Scalable Operating Systems

Wolfgang Schröder-Preikschat
German National Research Center for Computer Science
GMD FIRST
Rudower Chaussee 5, 12489 Berlin, Germany
wosch@first.gmd.de

Abstract

This work describes the design of scalable operating systems that have to meet the needs of various computer architectures and user applications. The focus is on how to organize and structure these operating systems and not a discussion of algorithms that implement specific system features. Thus, it is a software engineering course explaining operating-system structuring rather than implementation. The PEACE operating system is used as a case study in order to clarify the various design aspects considered.

1 Introduction

Operating-system construction is primarily a design and structuring problem. This is true for all kinds of operating systems and spans the areas of distributed, parallel, real-time, and fault-tolerant computing. The situation does not change with massively parallel (distributed-memory) architectures. Especially for these types of machines, a highly scalable system structure is required. This structure must scale well with at least hundreds to thousands of nodes, and should scale well with millions of nodes. Thus, the structure should meet the needs of both today's and future parallel architectures. In addition to all of these constraints regarding how to divide the complexity of the system, there is also the requirement of finding a system structure that guarantees the least possible message startup time for a given parallel application.

Scalability means the property of an operating system that is at least statically adaptable (at compile or binding time) to given user environments. This property does not only mean to be adaptable to expansive environments. That is, not only upward scalability but also downward scalability will be the characteristic of a scalable operating system. Upward scalability means *functional enrichment* and goes with increasing the system complexity and providing general system services. In contrast, downward scalability means *functional emaciation* and finds the extreme in highly specialized (i.e., customized) system representations of very small complexity.

First of all, operating-system scalability is a software engineering argument. Being dynamically adaptable at runtime is a technical improvement, calling for a statically adaptable operating system and specific system services that support dynamic operating-system restructuring. Scalability demands from the system designer to develop a clean, modular, and extensible and contractible software structure. One of the many aspects that must also be considered by the designer is to try to answer the question of whether microkernel technology is adequate for developing scalable operating systems or specific system software structures are required. If the latter is the case, how much will these structures differ from or have in common with (distributed) microkernel-based operating systems?

The purpose of this paper is to try to give an answer to this question. Section 2 opens with a brief discussion on scalability in microkernel-based operating systems. In Section 3, a well-known software engineering principle that helps to design scalable operating systems is presented. Afterwards, in Section 4, the design aspects are made clear by an in-depth case study. Section 5 concludes with some general remarks on current trends in operating-system design.

2 Microkernels and their Limitations

During the past two decades, operating-system research was largely dominated by, or (more or less) put on a level with, UNIX^{TM} research. The same applies to microkernels. Large efforts have been made developing a new technology through which UNIX functions appeared in a different shape. The era of monolithic systems came to an end. Mechanisms were developed aimed at migrating traditional operating system functions out of the monolithic kernel into user-level processes. So-called microkernels emerged. The appendage "micro" suggests that the kernel provides only a minimal function that allows user-level system processes to perform operating-system services efficiently. This function typically encompasses process management, inter-process communication, address space management and hardware (i.e., device) abstractions. However, it would be a wrong conclusion that microkernels are necessarily small in terms of code complexity [9]. The two dominant representatives of this new generation of operating-system technology, *Mach* [29] and *Chorus* [23], set a bad example when considering the software complexity of the kernel. Microkernels appeared on the (research/commercial) market the complexity of which is significantly greater than the complexity of the old-fashioned, monolithic and breakthrough bringing UNIX Version 6 [12, 13]. The latest example of this trend is WINDOWS-NT^{TM} [6].

When comparing microkernel-based systems to monolithic systems, a shifting of functionality took place but complexity remained almost identical. The complexity problem given with monolithic systems will not be solved by providing a (complex) platform that allows the designer to migrate functions from kernel to user space. Monolithic systems are not "clumsy" just because all of their functions are executed within the kernel address space. Rather, these systems are often inherently inflexible and difficult to maintain because of a poor software structure. UNIX Version 6 could have been an outstanding operating system had its software structure not been so bad. The same opinion applies to all its successors, including the systems of the BSD[1] branch. UNIX Version 6 is understandable not because of its software structure but because of its relatively small number of C code lines.[2]

The poor software structure and the incidental shortcomings (not only) of monolithic operating systems led, in the '70s, to the idea of *operating-system families* [10, 19, 20]. It seems that this software engineering idea has been largely ignored by the operating-system community. Operating-system families and UNIX were established during the same period. This fact may give an answer to the question of why UNIX has not been subjected to a family-based design. The family concept did not proceed beyond the early stages, while UNIX was already on the way to becoming one of the most commercially successful operating systems ever built. Since the late '70s, every new timesharing operating system has had to compete with UNIX. The success and failure of the new systems depended on the achieved enhancements and UNIX compatibility. Microkernel development has been (and still is) confronted with exactly this situation. Current microkernels can therefore be regarded as the best possible compromise to introduce new operating-system technology and, at the same time, maintain UNIX compatibility. However, they cannot be regarded as the ultimate approach to developing operating systems for any target market.

For example, custom-embedded or parallel systems play a slightly different game. All along, these systems make great demands on performance. Embedded systems operating in safety-critical environments also expect great things from the design methodology applied. The size of the entire system (in terms of memory space) is another important aspect. Operating systems in embedded systems usually reside in a PROM. UNIX compatibility is of minor importance, or incompatibility tolerated by the user, if performance, safety and size are all right. The rule of thumb was (and still is) keeping complexity as low as possible. A complex software system runs a very high risk of breeding inefficiencies and failing to meet safety checks. The concept of *program families* [18] was a meaningful contribution to avoiding these risks.

Similar considerations are true for distributed-memory parallel machines. Overloading processing nodes with unneeded microkernel functions does not only waste local memory resources but also limits scalability in general. OSF/1 AD, the microkernel-based parallel operating system for the Intel Paragon, gives an example of this circumstance. In many installations, the per-node memory consumption of OSF/1 AD is about seven to eight megabytes [7]. Thus, in a standard 16-megabytes node configuration, about half of the memory resources are confiscated by the operating system. Although OSF/1 AD implements virtual memory on every node, the user tries to partition the application program (i.e., data) such that it fits entirely into physical memory. Typically, the designer of numerical (parallel) programs is not willing to pay for the paging overhead of virtual memory systems, and therefore tries to arrange the programs in a way that minimizes, or even avoids, paging. Occasionally, program

[1] *Berkeley Software Distribution.*

[2] The entire Version 6 kernel consists of less than 10,000 lines of C code, including device drivers. Almost any computer science student is able to understand within one semester what happens inside the kernel. Only this aspect made UNIX a success story.

partitioning is done by the compiler.

The effect of the before-mentioned use pattern of parallel machines is that virtual memory degenerates to an unneeded microkernel function.[3] This fact was and is true for contemporary vector machines a la Cray and holds for (distributed-memory) parallel computers a la Intel.[4] Consequently, the modules implementing virtual memory introduce nothing but overhead. The overhead is twofold. On one hand, memory space is wasted due to unneeded code and data fragments. On the other hand, the message startup time is increased due to lower-level (microkernel) functions which assume that the message transfer takes place out of or into the virtual memory and are unaware of the specifically tailored user program that does not require the presence of virtual memory. For example, if the network hardware interface is able to perform the data transfer only from/to the physical address space, the driver will have to translate a logical memory address (pointing to the user-level message buffer) into a physical memory address. Depending on the underlying processor, this translation procedure may become a time-consuming task and thus introduces a communication performance bottleneck.

Unnecessarily, the scalability of user application programs for the Intel Paragon is limited mainly by the memory-consuming OSF/1 AD operating system, less by the amount of physical memory available on every node. Simply increasing the capacity of the physical memory of every node is not the solution. Besides scalability issues, compared with shared-memory parallel machines, the attractiveness of distributed-memory parallel machines lies in the favorable cost/performance ratio of a single node. More physical memory simply increases the hardware costs of a single node, and thus of the entire multinode machine.[5] As opposed to this, one can observe that the majority of parallel applications for distributed-memory machines get on with an operating system that exhibits a per-node memory consumption of about 50–500 kilobytes, even in the case of RISC-based system architectures such as the Intel Paragon. Thus, in its present shape, OSF/1 AD is entirely oversized for many parallel applications.

---

[3]Almost the same applies to multitasking support.

[4]Virtual memory on the processing nodes is in almost every case sacrificed, not only for technical reasons (such as the lack of a memory management unit). Data parallel programming of distributed-memory machines typically makes very large local address spaces in many cases superfluous. Moreover, an "unlimited" local address space is in contradiction to the MIMD principle, namely, to avoid the *von Neumann bottleneck* to the memory. Note that traditional virtual memory has little in common with virtual shared memory [11]. The latter was not introduced to solve the problem of memory overallocation, but rather to present a view of a global, common address space that is physically distributed. In particular, for performance reasons, this view can (and should) be implemented without relying on paging but rather on compiler and runtime system support.

[5]For example, upgrading the per-node memory from 16 to 64 megabytes does not necessarily imply that the hardware costs are only quadrupled. It depends on the actual state of a new-generation memory chip and how many chip providers of a new chip technology are available. Using prototype chips, the costs for 64 megabytes memory may be about 10 times as high as the 16 megabytes configuration. The cost factor may be about 5–6 if one can fall back on only one or two providers of the new technology. In the case of incompatibilities between the old and new chip technology employed, a complete redesign of the memory subsystem will be the consequence. This may increase the hardware costs more than a factor of 10.

Almost every new (commercial) operating system that appears on the market is based on a microkernel architecture [9]. The most recent example of this is WINDOWS-NT [6].[6] At first sight, this seems to suggest exploiting a microkernel as the common platform on top of which an operating system should be built. As was discussed above, along with OSF/1 AD, it is quite feasible to port even microkernel-based UNIX systems on distributed-memory parallel computers [30], with every node executing the microkernel, a subset of system servers, and at least one application task. However, the question arises as to whether this really is the right approach. Adapting portable operating systems to parallel machines and providing specific, portable operating-system support for parallel computing on these machines are two different matters. The adaptation of an existing operating system for use on a multinode parallel machine may fail to "*provide abstractions that are appropriate for certain applications*" and "*make efficient use of the hardware*" [26]. This observation does not only hold for the parallel computing area but is generally valid.

3 The Revival of Program Families

The approach for solving the problems discussed in the previous section is to understand an operating system as a *program family* [20] and to use *object orientation* [27] as the fundamental implementation discipline. The former concept (program families) helps prevent the design of a monolithic system organization, while object orientation enables the efficient implementation of a highly modular system structure. Both concepts support strongly the design and development of adaptable and, thus, scalable operating systems.

The program family concept distinguishes between a *minimal subset of system functions*[7] and *minimal system extensions*. It does not dictate any particular implementation technique. The minimal subset of system functions defines a platform of fundamental abstractions serving to implement minimal system extensions. These extensions, then, are made on the basis of an *incremental system design* [10], with each new level being a new minimal basis (i.e., *abstract machine*) for additional higher-level system extensions. A true application-oriented system evolves, since extensions are only made on demand, namely, when needed to implement a specific system feature that supports a specific application. Design decisions are postponed as long as possible. In this process, system construction takes place bottom-up but is controlled in a top-down (application-driven) fashion.

Representing an operating system as a program family is always meaningful, even if only a paper design is made. A clean and highly modular system structure is the natural outcome of this procedure. Maintenance of an actual system is significantly improved. Moreover, the provided system components are always tuned to a specific application. These components introduce overhead only with

[6] WINDOWS-NT is the largest microkernel ever built. It is hard, if not impossible, to make it understandable that microkernel technology results in a highly modular system structure when the platform proves to be all but "micro." This holds for all microkernel-based systems currently on the market.

[7] The notion of "*minimal basis*" will be used as synonym.

respect to their attempt at mapping a specific application onto the lower-level (abstract) machine. They are system specializations for a specific application rather than any type of application. Thus, the overhead in the mapping process will be the same as if the mapping was done by the application itself.

In its last consequence, applications become the final system extensions. The traditional boundary between application and operating system disappears. The operating system extends into the application, and vice versa. An incremental system design also promotes a dynamically alterable system structure.

Applying the family concept in the software design process leads to a highly modular structure. New system features are added to a given subset of system functions. Because of the strong analogy between the notions of "program family" and "object orientation" (Figure 1), it is almost natural to construct program families by using an object-oriented framework [5].

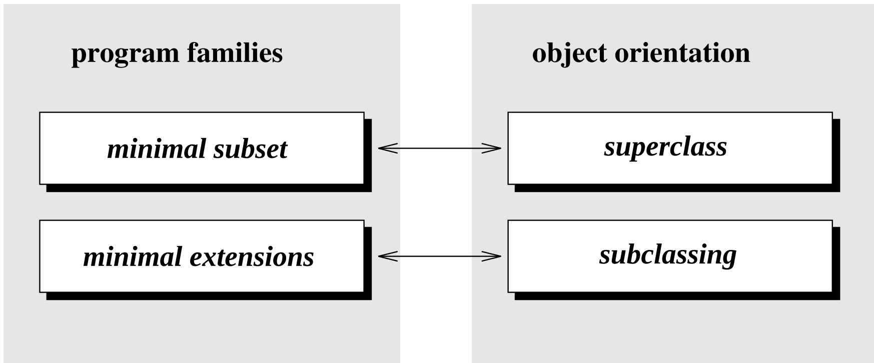

Figure 1: Program families vs. object orientation

Both approaches are, in a certain sense, dual to each other (i.e., exchangeable). The minimal subset of system functions in the program family concept has its counterpart in the superclass of the object-oriented approach. Minimal system extensions are thus introduced by means of subclassing. Inheritance and polymorphism are the proper mechanisms to allow different implementations of the same interface to coexist. Code reuse is significantly enhanced, increasing the commonalities of different family members:

> "We consider a set of programs to be a program family if they have so much in common that it pays to study their common aspects before looking at the aspects that differentiate them" [20].

For the development of (scalable) operating systems it is sensible to choose, above all, the program family concept as the fundamental design principle. Object orientation should primarily be used as an implementation, not a design instrument. Inheritance makes it much easier to implement and maintain an incremental system design, but is not mandatory for it.

Object orientation has many faces. In order to avoid misunderstandings, the "classical definition" [27] is employed. This definition says that objects are instances of classes and that inheritance can be applied to build class hierarchies. Classes describe *abstract data types* [14], the composition of which happens through inheritance. Thus, object orientation means:

$$\mathsf{abstraction} + \mathsf{classes} + \mathsf{inheritance}$$

Building hierarchies of classes also implies the specialization of their superclasses. Superclasses define fundamental object capabilities. Through inheritance, these classes are adapted to given application and hardware demands. In this way, specialized subclasses inherit and possibly extend the capabilities of their superclasses. Specialization differentiates between *single inheritance* and *multiple inheritance*. The former case always uses only a single superclass to compose the new level in the hierarchy (i.e., subclass), whereas the latter case does so with several superclasses [3].

Thus, a superclass may have many appearances, a circumstance that is described by the notion of *polymorphism*. Being virtually polymorphic is not only a capability of classes, but also of the methods defined by the classes. In this case, a superclass provides methods (functions or operations) that can be redefined by subclasses to achieve user customization. If instances (objects) of the specialized classes are created, *dynamical binding* (or *late binding*) takes place to associate the methods contained in subclasses with the superclass containing the original (deferred) definition. By applying the superclass scope, method invocation automatically leads to the invocation of the associated subclass method. This works transparently to the calling instance, namely, the object client.

An object-oriented system does not only consist of a very large number of objects. It must also provide "tunable" class abstractions. Above all, object orientation means *inheritance*, and thus specialization. Because of their customizability, object-oriented operating systems are open and application-oriented: they are scalable, at least from the software engineering viewpoint.

4 A Case Study

The following sections discuss the design of PEACE [25], explaining how the concept of program families can be applied to developing a highly modular and scalable operating-system structure. Originally designed for large, scalable (distributed-memory) parallel machines, PEACE is also used as a platform for distributed computing, real-time control, and education in computer science (particularly, operating systems).

Following the principle of *stepwise refinement* [28], the macrostructure of the system is presented first. This structure reflects the overall system architecture and deals mainly with splitting an operating system into individual distribution units. Afterwards, the microstructure of the system is presented. The question considered is how to design the minimal subset of system functions in order to meet the needs of a very large number of applications. The microstructure design focuses on performance and serves to implement a scalable operating-system platform for distributed-memory machines. In contrast, the macrostructure addresses the problem of how to divide system complexity by identifying loosely coupled but closely related building blocks.

4.1 Global System Structure

The maxim for the design and development of scalable/distributed operating systems is to reduce the number of site-dependent components to an absolute minimum. Hereafter, a site denotes either a cluster, a node, a processor, or an address space. Thus, the design must lead to an almost arbitrarily configurable system structure. Whether a system component is sharing a specific site with other system components should be a matter of configuration and not implementation. A highly modular system structure is required. This structure must enable, but not enforce, the decoupling of functional units such that a high degree of decentralization becomes possible.

In the PEACE philosophy, every family member is constructed from three major building blocks (Figure 2). These building blocks are the *nucleus*, the *kernel*, and POSE, the *parallel operating system extension*. In addition to the system components, the *application* is considered as the fourth integral part of this architecture. The application largely determines the complexity of a family member and, for example, the distribution of the building blocks over the nodes of a parallel machine.

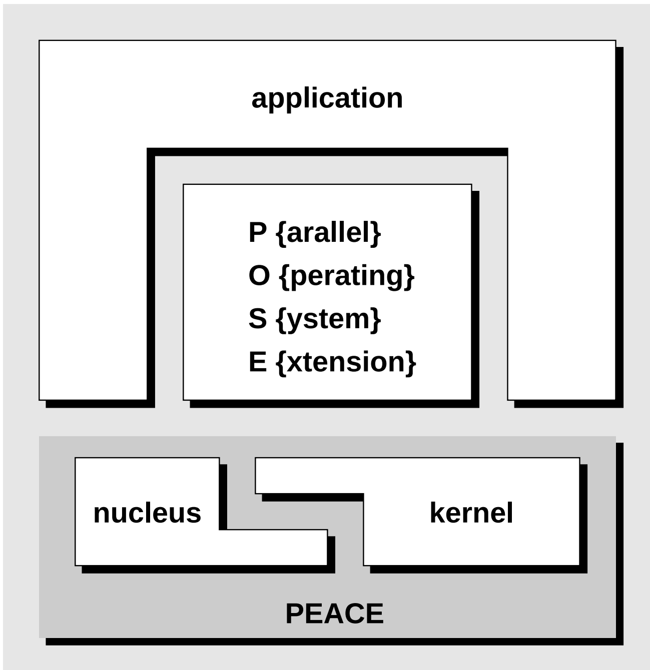

Figure 2: Building blocks

The nucleus implements systemwide inter-process communication and provides a runtime executive for the processing of threads. It acts as the minimal basis and is part of the kernel domain, with the kernel being a multithreaded system component that encapsulates minimal nucleus extensions. These extensions implement device abstractions, dynamic creation and destruction of process objects, the association of process objects with naming domains and address spaces, and the propagation of exceptional events (traps and interrupts). Application-oriented services such as naming, process and memory management, file handling, I/O, load balancing and (inter-) networking to provide some host access are performed by Pose.

The kernel and Pose services are built by active objects implemented by lightweight processes. In contrast, the nucleus is an ensemble of passive objects that schedule active objects. Kernel and Pose are *multithreaded entities*, and thus support concurrent service processing. An entity provides a common execution domain for active objects and is considered as the *unit of distribution*, while an active object is considered as the *unit of execution*.

The nucleus and kernel constitute the *kernel entity*. This entity defines an abstract processor for the execution of possibly multithreaded tasks. The sole purpose of the kernel entity is to provide hardware abstractions that make (parallel) applications and Pose independent from the physical characteristics of a given processing node.

4.1.1 Operating-System Scalability

Entities are system extensions. They are loaded on demand and (in most cases) can be arbitrarily distributed. Loading a kernel entity on demand means bootstrapping a node. The need for having a specific entity running on a particular node is expressed by either the application or a system administrator. In the former case, the entity is loaded when the service it represents is called for the first time. The entity vanishes when this service is no longer needed. This may happen, for instance, upon termination of the application entity that demanded the service. Note that in this model, application entities are not distinguished from system entities such as the file manager, process manager, or external pager. In other words, provided the application is properly structured, incremental loading of parallel/distributed applications works exactly the same as incremental loading of the operating system. Hence, it is free of charge.

In the case of the system administrator, an initial configuration description specifies which entities must be initially loaded onto which nodes when bootstrapping. This procedure loads "opaque" entities from some bootstrap device (usually a disk or network) and stores them into the local memories of the nodes. Understanding parallel/distributed programs as groups of related entities, bootstrapping of these opaque entities can be used to get applications, not only system components, running. The purest form of batch processing is thus achieved.

Either way, Pose is a dynamically alterable building block. The design specifies that Pose can alter its shape depending on the applications that are to be executed by the machine. Thus, applications determine the actual complexity of Pose. In consequence, they are responsible for bringing (i.e., bootstrapping) the operating system on the machine.

Incrementally loading the operating system does not mean placing Pose entities on the node where the demanding application is residing. Since these entities may be qualified as site-independent, they could also be loaded on any appropriate node. In order to achieve an "optimal" mapping, expressing some nearest-neighbor relationship between the entity and either a higher-level application or a lower-level hardware/software component may be sensible. This is where the configuration aspect comes in. The entity must be attributed properly, and the attributes must be interpreted by some low-level load-balancing service.

4.1.2 Kernel Scalability

Logically, both the applications and POSE are assumed to be executed in the nonprivileged processing mode (*user mode*) of the underlying processor. In contrast, the nucleus and kernel are designed for execution in the privileged mode (*supervisor mode*). Thus, the logical design defines an artificial boundary isolating these two components from other entities. This does not mean that the actual system representation must always consist of isolated system components. Rather, the purpose of introducing the boundary is to guarantee downward scalability.[8] Therefore, it is of importance to clearly identify those components that, under certain circumstances, must be executed in the privileged mode and others that can be executed in the nonprivileged mode. Note that supervisor-mode components also constitute the set of (logically) node-bound entities. By designing a system structure that exhibits a "vague" boundary between the user and supervisor modes, one is able to determine the number of node-bound entities dependent on the application that must be supported.

The design defines a *family of parallel operating systems* that is extended by a *kernel family*. The kernel family contains members that mainly distinguish between the separation of the user and supervisor modes. Basically, these members implement a number of different operating modes. The members are all node-bound, and therefore classified as site-dependent. The operating-system family is composed of site-independent entities. It is a configuration matter as to whether or not these entities share the same site (i.e., node, processor or address space) with the kernel and the nucleus. Similar considerations hold for application entities.

This design especially promotes address-space sharing between all entities of the same node, provided that the structure of the application allows it. In a single-tasking mode of operation, the application, POSE, kernel and nucleus all may execute within a single address space.[9] However, it must be clear that only in the presence of secure programming languages and the absence of temporary node failures can the model of a single address space be considered safe. Nevertheless, this model is still applicable for mature applications—and, of course, for mature system components—to prevent per-node overhead in crossing "artificial" subsystem boundaries.

Whether the application or POSE must cross address space boundaries to invoke the nucleus depends on the implementation of the calling sequence. The

[8] The system design must be complete in order to be able to later integrate a version of a system component of significantly less (i.e., scaled-down) functionality.

[9] How big the resulting (nonsequential) program then becomes depends exclusively on what the application demands from the system.

service access point of the nucleus is represented as an abstract data type with two different implementations: an *intra-domain call*, with the callee sharing with the nucleus both the same node and address space segment, and an *inter-domain call*, with the callee sharing with the nucleus the same node but not the same address space segment.[10]

4.2 Decentralization of System Components

The functional hierarchy defined between POSE, the kernel and the nucleus makes possible a very high degree of decentralization (Figure 3). All components are encapsulated by (active/passive) objects, so that object invocation schemes are required to request service execution. Well-defined interfaces enforce a clean separation of all building blocks. Different styles of calling sequences to cross the (nucleus, kernel and/or POSE) interface enable various system configurations. These configurations particularly distinguish between a single or multiple address space model being implemented on a node.

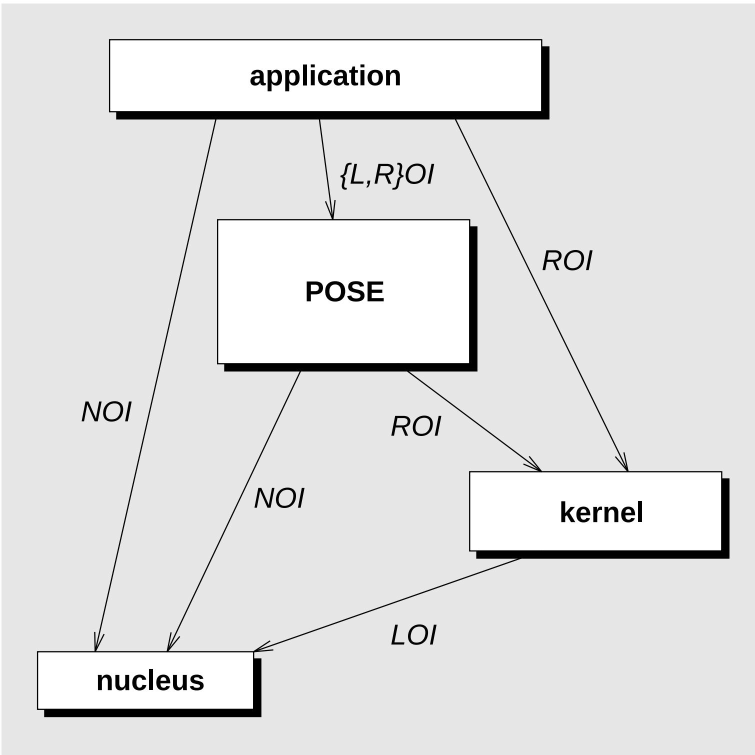

Figure 3: Problem-oriented invocation schemes

Nucleus services are made available to the application and POSE via *nearby object invocation* (NOI). The logical design assumes a separation of the nucleus from the application (and POSE), which may call for the use of traps to invoke

[10]An inter-domain call is also denoted as a cross-domain call.

the nucleus and for address space isolation. This is the place where cross-domain calls may happen. The nucleus is "nearby" the using entity. It shares the same node, but not necessarily the same address space segment, with the entity. Since the kernel and nucleus together reside in the same address space, the kernel performs *local object invocation* (LOI) to request nucleus services. LOI causes the thread to continue execution within the same address space (or domain). In contrast, NOI causes the thread to continue execution within a different address space on the same node.

Logically, kernel services are made available via *remote object invocation* (ROI). The ROI scheme always implies context switching, but not necessarily address-space switching [17]. A separate thread of control is used to execute the requested method. In contrast, NOI implies the activation/deactivation of the nucleus address space via local system call traps. The implementation of ROI takes advantage of the networkwide message-passing services provided by the nucleus, and thus is logically based on NOI. However, since the nucleus and kernel together define a single program, nucleus primitives invoked by the kernel to either issue or accept ROI requests hence will not be activated via NOI but LOI.

Services of POSE are requested via LOI and ROI. The former scheme is used to interact with the POSE runtime system library, whereas the latter is used to interact with POSE active objects. In certain situations the POSE library directly transforms a LOI into one or more ROI requests to the kernel. The POSE service, then, is not provided by an active but by a passive object (i.e., on a library basis).

The NOI and ROI patterns are nothing but specific RPC variants. Both implement a synchronous control flow transfer via a "narrow channel" [16]. For example, the narrow channel is represented by a trap handler (in the case of NOI) or the message-passing system (in the case of ROI). Thus, NOI and ROI indicate the two technical means of carrying out an object-bound RPC. In both cases, the control flow transfer entails logically changing protection domains (e.g., address spaces or segments). However, different from NOI, ROI causes a (lightweight) process switch. NOI forces the current thread of control to continue processing in a different domain; that is, NOI corresponds to an object-bound *lightweight RPC* [2].

4.3 Actual System Structure

From the design point of view, neither the kernel nor POSE need to be present on every node, only the nucleus. In a specific configuration, the majority of the nodes of, for example, a (distributed-memory) parallel machine are equipped with the nucleus only. Some nodes are supported by the kernel, and a few nodes are allocated to POSE. All nodes can be used for application processing, but they are not all obliged to be shared between user tasks and system tasks. The functional hierarchy of the three building blocks expresses the logical design of PEACE, but not necessarily the physical representation. However, it depends on the actual operating-system family member whether the various schemes of object invocation (Figure 3) become effective as specified by the design or can be replaced by a simpler, more efficient alternative.

Although the functional hierarchy assumes NOI for the interaction between application (POSE) and nucleus, the LOI scheme is used for those members of the kernel family that place their focus on performance. The entrance to the nucleus is represented as an abstract data type with two implementations. The first implementation sacrifices vertical and horizontal isolation. Thus, there is neither a separation between the user and supervisor modes of operation (vertical isolation) nor a separation between competing tasks (horizontal isolation). In this case, NOI actually means LOI. Horizontal isolation means that user/system entities have a private address space, that is, operate in a private protection domain. The second implementation assumes complete (i.e., vertical and horizontal) isolation and requires a trap-based activation of the nucleus. NOI then becomes a cross-domain call. The variants basically distinguish between single-tasking (no isolation) and multitasking (isolation) modes of operation. They implement different members of the kernel family.

A single-tasking mode of operation implies that only a single address space is supported on the node. As pointed out above, NOI then takes the form of LOI. This also means that all interactions with the local residing kernel happen via LOI too. Thus, instead of ROI, as specified by the design (see Figure 3), the more efficient LOI alternative is applied in that case. Note that, in this case, no address space boundary exists between higher-level entities and the nucleus. Since the nucleus is part of the kernel address space, sacrificing nucleus isolation also implies sacrificing kernel isolation. As the kernel "uses" [19] the nucleus, and thus depends on its correctness, it makes little sense to isolate the kernel in this situation. Consequently, locally invoking the nucleus or kernel in the single-tasking configuration causes no context switch.

In the single-tasking configuration, the kernel is of *hybrid structure*. Locally issued requests are handled on an LOI basis as mentioned above. However, since there might be remotely residing system components requesting the allocation of local kernel resources (e.g., enabling the interaction with devices attached to the node), kernel services must be provided via ROI also. In other words, there are different invocation schemes existent at the same time to interact with the kernel.

4.4 Microkernel Family

The PEACE nucleus implements a minimal subset of system functions and is a compromise between scaling transparency and efficiency. The compromise was made with respect to the interface specification of the abstract data type which represents that subset. The internal behavior of this data type is manifold and basically distinguishes between the single- and multitasking mode of operations. The functions provided by the nucleus and specified by the interface of the abstract data type(s) define a "minimal but perfect" subset. It is *minimal* because it does not anticipate any particular design decision that would be a handicap for the implementation of higher-level user/system abstractions. It is *perfect* because it already supports a large number of applications.

Offering dedicated single-tasking and multitasking kernel implementations enables the user to deal with the trade-off between performance and functionality on an individual basis. Applications whose tasks can be mapped in a one-to-one correspondence with the nodes are not punished by multitasking overhead, because these applications can be supported by a single-tasking kernel. Experiences have shown that this overhead may take up to 74% of the time used to execute a user-level (networkwide) *send-receive-reply* sequence [24]. Table 1 shows how the 74% multitasking overhead is distributed over typical microkernel functions. Note that a state-of-the-art microkernel, however, supports only multitasking, and thus unnecessarily draining computing power from a single-tasking application.[11]

Multi-Tasking Overhead	
Operation	*Proportion*
inter-domain calls	5%
vector unit housekeeping	8%
task scheduling	14%
interrupt handling	22%
packet movement	25%
total	74%

Table 1: Message startup time problem

The minimal subset of system functions is represented by a *nucleus family* that implements four different modes of operation (Figure 4).[12] Each mode is represented by a subfamily. The entire family tree shows different nucleus versions, with the root (top) the simplest and the leaf (bottom) the most complex instance.[13] As functionality and complexity increases, overhead, and thus communication latency, increases.

This organization of the nucleus, and hence the kernel entity, is one of the major differences from state-of-the-art microkernel designs. The kernel entity defined is not a single microkernel, but a "*microkernel family.*" It can be adapted to the individual needs of parallel and distributed applications by providing family members that exhibit different system characteristics regarding functionality and performance.

[11]For the SUPRENUM implementation of PEACE, which is based on a 2-MIPS MC68020 processor, systemwide bidirectional inter-process communication using a 64-byte packet takes about 784 μs in the multitasking mode and 204 μs in the single-tasking mode.

[12]Also referred to as "*kernel family*" or "*microkernel family.*" Nevertheless, the notion of a "nucleus family" is more appropriate since the variety of the nucleus implementation causes the different operating modes of a node. In contrast, depending on the supported operating mode, kernel service functions are either present or missing. For example, multitasking requires kernel functions to create and destroy address spaces. These functions are not present when the nucleus implements single-tasking.

[13]The arrows in Figure 4 indicate an inheritance relationship between the members of the nucleus family. However, not classes (or objects), but functions, are set in relation to each other. The nucleus family tree shows the functional relation between family members.

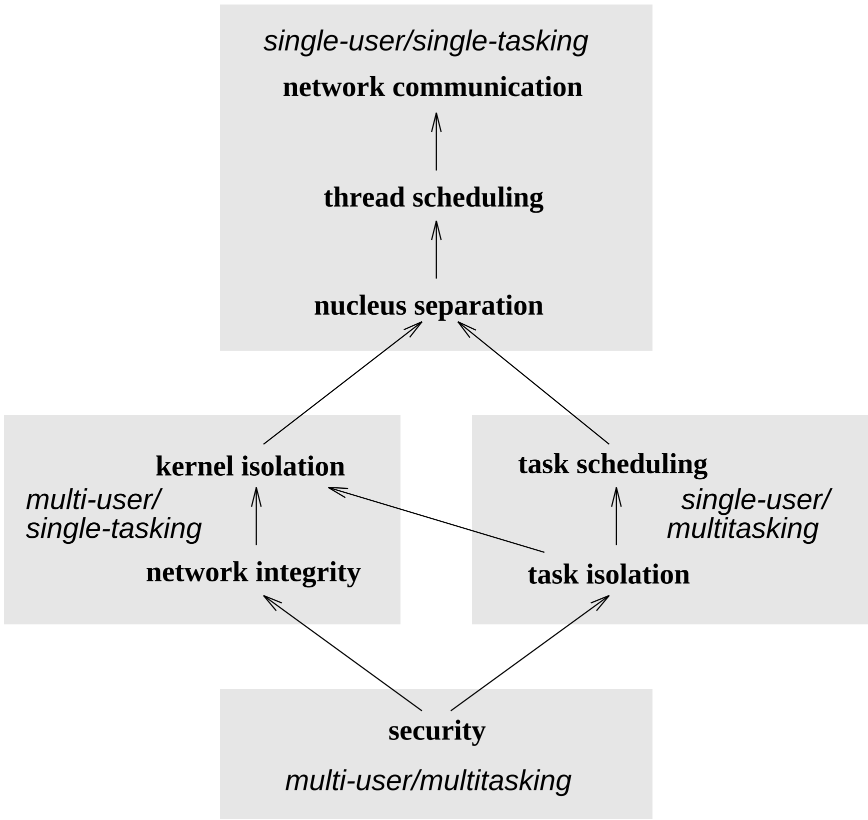

Figure 4: Nucleus family tree

4.4.1 Single-User/Single-Tasking

The root of the nucleus family tree shows three different members supporting the single-user/single-tasking mode of operation, with tasks possibly being multithreaded. This mode of operation allows for only one user application at a time and merely supports the processing of one task per node. On every node, a single-address-space model is implemented. Consequently, the entire machine is subjected to batch processing. This enables the most efficient execution of the user program. At the same time, it may lead to the most inefficient utilization of the machine, since the machine cannot be shared between a number of applications.

Minimal support for distributed applications means providing *network communication*. This feature is provided by the most fundamental member of the nucleus family. At this stage, thread dispatching is nonpreemptive and completely under the control of the application task. This nucleus representation is extremely lightweight and supports the notion of *featherweight processes*. Featherweight processes are a scaled-down version of lightweight processes. Each of them implement the purest form of a unit of execution, without any consideration of protection and security measures. NOI is technically represented as

LOI.

Thread preemption is the minimal extension to network communication. This results in *thread scheduling* on a timer basis, actually introducing a second scheduling level. The thread-scheduling member implements CPU protection [22], since a single thread (or task) is no longer able to monopolize the processor. The additional level knows the *bundle* as the scheduling unit and allows safe integration of kernel-level lightweight processes that implement the service access point for ROI. These threads are then combined into a *kernel bundle*. All other threads form the *application bundle*. Since bundles are subjected to preemptive scheduling, kernel service processing works independent from the processing of the application task. Consequently, threads no longer execute under the control of the application task but of the system (i.e., nucleus). This ensures that remotely requested services are processed the latest, when the thread scheduler has gained control and selected a clerk for execution—provided that the application task sharing the same address space with the kernel entity is "well-behaved."

The next nucleus family member does not really introduce any additional functionality but provides the platform for the multitasking and/or multi-user mode of operations. These modes of operation call for isolation and protection measures. Thus, the *nucleus separation* member introduces NOI and ROI patterns as minimal extensions to thread scheduling in order to request services from the nucleus and the kernel, respectively. This kernel family member relies on inter-domain calls to invoke the nucleus.

Higher-level entities are physically uncoupled from nucleus code. Since every nucleus and kernel takes the form of an abstract data type, these entities are also logically uncoupled from kernel-level data. This supports the dynamical change of the operating mode of a node by replacing the actual kernel entity with a different implementation. However, note that only code separation is supported, but not memory protection. As a consequence, the passing of complex data structures between the kernel entity and higher-level entities is still straightforward and involves no MMU programming.

4.4.2 Multi-User/Single-Tasking

In a distributed-memory parallel machine, the multi-user mode of operation is feasible even if only a one-to-one correspondence between tasks and nodes is supported. The entire multinode machine can be allocated to different users at the same time. For this purpose, different *user partitions* are provided, with every user application executing within a private partition. This does not require local ("on-board") security measures to protect the tasks from each other, but it requires network interface protection. With direct network access, the user task would be able to intrude into the network, thus into tasks mapped to different user partitions. In principle, this violates the integrity requirements of each of the user applications.

In order to provide the multi-user function, the kernel entity must be entirely isolated. Memory protection is introduced as a minimal extension to nucleus separation. This yields a new family member that provides *kernel isolation*.

Because the nucleus is part of the kernel entity, applying memory protection to the nucleus also implies the isolation of (at least parts of) the kernel address space. Vertical isolation is applied to protect the kernel entity. In this case, nucleus invocation (i.e., crossing the trap interface) does not usually increase the overhead. However, the passing of data structures becomes more heavyweight. It mainly depends on the MMU whether or not additional overhead is introduced. In any event, the increase of nucleus functionality is accompanied by a potential loss of communication performance.

Once the kernel entity is protected, tasks are no longer capable of intruding into lower-level software components thus bypassing protection domains. However, in order to communicate, tasks need to know communication endpoint (i.e., thread) identifiers. These identifiers must be passed down to the nucleus along with a communication request. In a multi-user configuration, tasks mapped to different user domains must therefore be prevented from being able to successfully use communication endpoint identifiers of other domains. This is the purpose of the minimal extension to kernel isolation, called *network integrity*.

With network integrity, communication firewalls between different user applications are introduced. Every user application builds a unique communication domain. The same holds for the set of system processes that constitute the operating system. Within the same domain, communication is unlimited. In order to invoke system services, however, application processes must communicate with system processes. The changing of communication domains must be possible. This requires different communication domains to overlap in order to enable communication. Hence, communication security does not mean complete isolation, but rather controlled access.

To model and enforce protection domains in the above-mentioned sense, a capability-based approach [8] proves to be the appropriate solution. This approach grants object access only if a thread (i.e., subject) has possession of that object or one of its proxies. An object must be created and bound to a systemwide unique identifier before it can be used. It is assumed that a unique identifier cannot be deduced [15]. In order to achieve this, the nucleus generates a random number which, combined with a global hash key, is used to make identifiers systemwide unique. Note that this procedure works autonomously and need not be controlled by a central system component. Once instantiated, the creator implicitly possesses the new object and henceforth acts as its owner. The owner has unlimited access rights to that object. It is the autonomous decision of the owner to make the object either globally known, by exporting its unique identifier, or even accessible, by exporting its *proxy object* [17]. Thus, the access domain of an object may be extended only by the owner. Consequently, the nucleus need not verify that communication to an (active) object is allowed but only whether this object exists—knowing the capability of this object enables communication. Verification of the existence of an object then means checking for the validity of the systemwide unique identifier of the object. This means that the nucleus only has to deal with these unique identifiers and not with the object actually being communicated with.

4.4.3 Single-User/Multitasking

The scheduling paradigm introduced by the thread-scheduling member does not distinguish between multiple autonomous program entities. Mere execution of a single application program (i.e., task) is supported. Having introduced nucleus separation, the processing of several application programs becomes possible by slightly extending the scheduling functions. Note that the nucleus can easily be shared by a number of application tasks. It exploits either a trap mechanism or a statically known "jump table" to provide a common entry point for independently generated programs.

A task is represented as a *team* of lightweight processes, with the team possibly exhibiting several thread bundles that can be autonomously scheduled. The team owns resources such as message buffers, memory space for stacks and process descriptors; *task scheduling* then means keeping track of the resources owned in order to select the next task ready to run. A scheduling discipline that takes into account the current load and thus tries to improve global system utilization may be favorable. This further calls for minimal scheduler extensions to those already provided by nucleus separation.

At this stage, multitasking can be readily supported even if private address spaces are not provided. A private address space serves two basic purposes. On one hand, it implements memory protection, isolating programs from each other. On the other hand, it defines a logical address space in which to execute a program, enabling code/data relocation at runtime. Being relocatable is also a property of *position-independent* code, which then must to be generated by a compiler. In addition, the use of secure programming languages supports program isolation without the necessity of address space protection hardware.

For all these reasons an intermediate step is made, introducing the task-scheduling member as a platform for multitasking. A minimal extension to this platform is introduced with *task isolation*, which encorporates vertical and horizontal isolation. This extension is used to generally improve system availability and in those cases where neither the programming language nor the compiler provides adequate support for a single-address-space multitasking mode of operation. Note that task isolation extends not only task scheduling but also kernel isolation. The latter family member already provides vertical isolation to protect the kernel entity from higher-level user entities. With task isolation, the nucleus is also functionally enriched by measures of horizontal isolation to protect user entities from each other.

4.4.4 Multi-User/Multitasking

By combining the functions of network integrity and task isolation, a new nucleus family member emerges. This member introduces multi-user *security*, leading to the fourth operation mode supported by the kernel family. Multi-user security in a distributed environment demands communication firewalls and protected address spaces. The former is achieved by ensuring network integrity and the latter by enforcing task isolation. The peak of system functionality has been reached; however, only at the cost of system performance.

4.5 The Role of Naming

All sorts of system services provided by the POSE, nucleus and kernel are encapsulated by one or more objects. The logical design assumes that only the nucleus provides services by means of *passive objects*. POSE and the kernel are logically built with *active objects*; however, both may exhibit an actual structure in which passive objects are used for system composition as well.

In the case of an active-object-representation, interaction with the services happens via message passing. Systemwide unique identifiers specify the communication endpoint address of a message transaction. They are numeric, structured low-level identifiers possessing a location-dependent value. This representation is used mainly for performance reasons. The low-level addressing scheme, however, has a drawback for network transparency in general and operating-system scalability in particular. On the other hand, it relieves the nucleus (or kernel) of keeping track of local communication gates (i.e., forwarding identifiers) that point to remote tasks or servers. In order to maintain transparency and scalability, a logical address of a service-providing active object is required. The address must not include any location-dependent information. A naming scheme is required that performs the mapping of logical to physical object addresses. Figure 5 illustrates the PEACE naming scheme.

Symbolic identifiers, represented as user-defined character strings, introduce location transparency. Moreover, they are applicable not only to address active objects, that is, objects which are capable of actively receiving messages. A symbolic identifier may refer to any kind of object. Thus, it serves as a general addressing mechanism. Symbolic identifiers are stored in *name spaces* and associated with systemwide unique identifiers. Name servers will take care of the mapping of symbolic to systemwide unique identifiers: they are used to map logical server addresses onto physical server addresses. As shown in Figure 5, this mapping must reflect various aspects that deal mainly with the modeling of a family of operating systems. The following subsections describe the mappings shown in Figure 5 in detail.

4.5.1 Access and Location Transparency

The nucleus operates with systemwide unique identifiers for performance reasons. In addition, the nucleus implements the minimal subset of system functions that must reside on every node of networked computer system. This is the starting point of the following considerations—and what the mapping process is aimed at.

With the decision to implement system services using ROI (or RPC), *access transparency* is introduced. The user sees only a procedural interface. There is no way to apply a systemwide unique identifier unless demanded by the interface of the service function. In principle, this leaves it solely up to the ROI layer to cope with "physical server addresses." Thus, a good design makes sure that the service interface specification does not contain physical object addresses.

The use of symbolic identifiers to name system components that provide services hides the presence of systemwide unique identifiers. The logical object

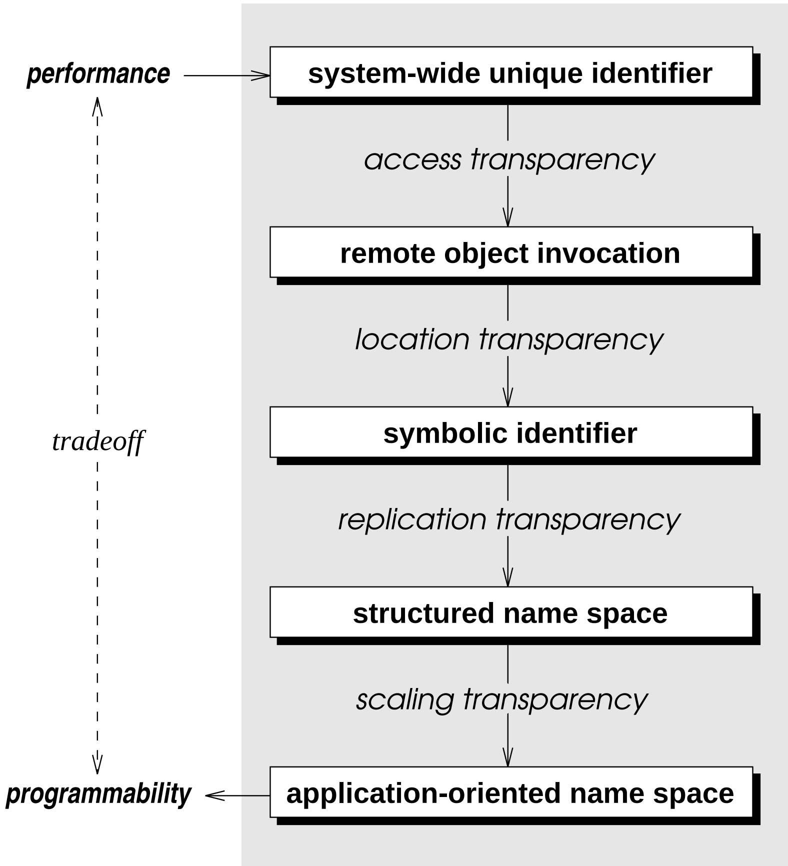

Figure 5: Naming abstractions

addresses introduce *location transparency*, which is achieved by the dynamic mapping applied to obtain systemwide unique identifiers from symbolic identifiers. At this level, naming and communication exceptions are handled. If the use of a stale communication endpoint address has been signaled by the nucleus, the user-level naming subsystem is requested to resolve a symbolic identifier. The result is the actual communication endpoint address of an active object.

A symbolic identifier is a name that does not include "hard-wired" address information. In particular, the naming system does not interpret the character strings, and therefore cannot benefit from encoded site information. Site information may be encoded in these strings but will have no effect on the mapping. Symbolic identifiers are only significant end-to-end. They are interpreted by either the client or server. A name server takes the character string, applies the mapping function and thus returns the associated systemwide unique identifier. In reality, even a name server does not know anything about systemwide unique identifiers. Rather, it maps a character string onto a byte string, whereby the byte string in most cases represents a bit pattern that stands for a systemwide unique identifier. Thus, both input and output data of a name server are completely user-defined, with name servers implementing "*associative memory.*"

4.5.2 Replication Transparency

The naming approach offers an interface to active objects by which only location-independent symbolic identifiers are supported. The motivation is to make active objects configurable by a third party. Given a site-independent active object, the third party decides where to place it. The naming system does not benefit from site-dependent information, such as the node address of an active object encoded into a symbolic identifier. As a consequence, replicated and code-identical active objects cannot be distinguished by name. All these objects could use the same symbolic identifier (i.e., character string) to make themselves globally known. This issue is revisited below. The potential for name clashes due to code-identical instances must be solved by means of a structured name space.

The global name space is partitioned into a number of *name planes.* Each name plane implements a unique naming domain within which a single symbolic identifier is guaranteed to be unique. The same identifier may result in different outputs of the mapping function (when all name planes are taken into account). A structured name space of this sort (i.e., relying on different name planes) introduces *replication transparency.* The third party, for example, associates active objects with name planes and thus, logically, with individual naming domains. Objects can be migrated to different name planes, extending or reducing the scope. Active objects may or may not share the same name plane. For instance, tasks of the same application will be bound to a single name plane, to enable network-transparent object localization. As a consequence, tasks belonging to different applications cannot identify each other. This is a prerequisite for maintaining multi-user security.

4.5.3 Scaling Transparency

A family of operating systems implies giving every application, logically, its "private" ensemble of server objects. This not only means providing different kinds of services, such as process management for one and file management for another application. It also includes providing different implementations of the same service interface. Servers are abstract data types, their interface being described by an "*abstract class.*" The class name is then the almost natural choice for use as the symbolic identifier to address a given instance of that class.

For these reasons, a number of different objects may be bound to the same logical name. Using name planes helps to reestablish the uniqueness of each name. However, this alone does not solve the problem that a given application must have access only to those name planes that are keeping track of (server) objects belonging to the application domain. The name space must be structured in an application-oriented manner. Name planes are to be configured on a per-process basis. This introduces *scaling transparency*, that is, it supports a scalable operating-system architecture.

5 Concluding Remarks

Scalability means the property of an operating system that is at least statically adaptable (at compile or binding time) to given user environments. Being dynamically adaptable at runtime is a technical improvement, calling for a statically adaptable operating system and specific system services that support dynamic operating-system restructuring. Scalability demands from the system designer to develop a clean, modular, and flexible software structure. The concept of program families provides the proper design framework and object orientation is the adequate implementation discipline. Thus, at first, operating-system scalability is a software engineering argument.

There are only a very few operating-system examples, not to say "exceptional cases," that are family-based designs. Modern representatives are *Choices* [4] and PEACE. The family concept is a means to an end but not an end in itself. It requires that the designer be very disciplined and precise, if not pedantic, in the construction of operating-system software. Typically, an operating-system family exhibits a highly modular and hierarchical software structure. A system with tens of levels is not uncommon. The hierarchy is built by postponing design decisions that would restrict the already designed family [10]. A major concern is to decide which functions to exclude from a design. In other words, functions are introduced only on a per-application basis (i.e., if demanded by a higher layer). The new functions define an off-spring of the existing family.

Always forcing oneself to reason about the necessity of a certain function at a certain level is often deemed too academic, less pragmatic, causing an overstructuring of the resulting system—but it is worth it. The self-assessment during the design (and implementation) process helps to reduce the complexity of lower-level software components to an absolute minimum. Functions are not simply added to an existing compilation unit when they are not used by other functions of that unit. This rule holds even when new functions share the same knowledge (e.g., data structure) with the functions contained in the compilation unit. Similarly, the implementation of an *abstract data type* (ADT) must not necessarily mean ending up in a single compilation unit. In order to maintain the family idea, a restructuring of the existing ADT may be the consequence. The outcome of restructuring would then be that the new and old functions share the same knowledge but are not forced to coexist. This brief discussion hints that the implementation of a program family entails a large number of ADTs, compilation units or files.

Applied to the microkernel concept, a family-based design would imply providing the above-mentioned functions, such as process management, interprocess communication, address space management and hardware (i.e., device) abstractions, only on a per-application basis. In particular, the multiuser/multitasking capability of microkernels would have to be provided only by a dedicated family member, not all of them. Current microkernels are not designed this way. Specific hardware developments become necessary to help microkernels execute applications more efficiently and, more precisely, to allow applications to bypass the microkernel. It is mandatory for an operating-system designer to first seek *lightweight software structures* before calling for faster hardware to fulfill the performance requirements.

Current microkernels are not lightweight enough to support a broad range of applications. This situation induced diverse groups to invent the notion of the "*nano-kernel*." An example from the parallel operating system area is *PARIX* [21]. However, even this attempt to express the light- or even featherweightness of microkernels is somewhat misleading. It does not really overcome the original problem, providing a computing platform that scales well not only with the underlying hardware but also with the given application. In many cases, the nano-kernel is nothing but a threads package running on the bare hardware and providing dedicated communication primitives. Multiuser/multitasking is not supported. Even the nano-kernel presents only a single solution to a problem space that requires a number of solutions. It does not scale up, while the microkernel does not scale down. A *microkernel family* would include both.

The *pico-kernel* used in the PANDA [1] approach shows that microkernels and even nano-kernels are not the last resort. With operating-system kernels getting smaller and smaller—the *femto-kernel* will be not long in coming—, programming language and compiler support are of steadily increasing importance. As PANDA demonstrates, designing an extremely scaled-down and efficient kernel does not necessarily result in a user-friendly interface. However, the question is what user-friendliness means. From the viewpoint of the kernel designer, user-friendliness means that applications are not hampered by inappropriate kernel abstractions. From the viewpoint of an application designer, user-friendliness often means easy-to-use kernel functions. In order not to lose all the advantages (in terms of efficiency and flexibility) and hide the peculiarities of extremely scaled-down kernels, compiler and/or programming-language support is required.

References

[1] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, and R. Schwarz. The PANDA System Architecture—A Pico-Kernel Approach. In *Proceedings of the Fourth Workshop on Future Trends of Distributed Computing*, pages 470–476, Lisbon, Portugal, September 1993. IEEE Computer Society Press.

[2] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1):37–55, 1990.

[3] G. S. Blair, J. J. Gallagher, and J. Malik. Genericity vs Inheritance vs Delegation vs Conformance vs... *Journal of Object-Oriented Programming*, 2(3):11–17, 1989.

[4] R. Campbell, G. Johnston, and V. Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *Operating Systems Review*, 21(3):9–17, 1987.

[5] J. Cordsen and W. Schröder-Preikschat. Object-Oriented Operating System Design and the Revival of Program Families. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems (I-WOOOS '91)*, pages 24–28, Palo Alto, CA, October 17–18, 1991.

[6] H. Custer. *Inside WINDOWS-NT*. Microsoft Press, 1993.

[7] R. Esser and R. Knecht. Intel Paragon XP/S—Architecture and Software Environment. In *Proceedings of Supercomputer '93*, Lecture Notes in Computer Science, Mannheim, Germany, June 24–26 1993. Springer-Verlag.

[8] R. S. Fabry. Capability-Based Addressing. *Communications of the ACM*, 17(7):403–412, July 1974.

[9] M. Gien. Micro-kernel Architecture – Key to Modern Operating System Design. Technical Report CS/TR-90-42.1, Chorus systèmes, Paris, 1990.

[10] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[11] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, 1986.

[12] J. Lions. A Commentary on the UNIX Operating System. Technical report, Department of Computer Science, The University of New South Wales, 1977. Second Printing.

[13] J. Lions. UNIX Operating System Source Code Level Six. Technical report, Department of Computer Science, The University of New South Wales, 1977. Second Printing.

[14] B. H. Liskov and S. Zilles. Programming with Abstract Data Types. *SIGPLAN Notices*, 9(4), 1974.

[15] S. J. Mullender and A. S. Tanenbaum. The Design of a Capability-Based Distributed Operating System. *The Computer Journal*, 29(4):289–299, 1986.

[16] B. J. Nelson. Remote Procedure Call. Technical Report CMU–81–119, Carnegie-Mellon University, 1982.

[17] J. Nolte. Language Level Support for Remote Object Invocation. Arbeitspapiere der GMD 654, Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin, Germany, June 1992.

[18] D. L. Parnas. On the Design and Development of Program Families. Technical Report BS I 75/2, TH Darmstadt, 1975.

[19] D. L. Parnas. Some Hypotheses About the "Uses" Hierarchy for Operating Systems. *Transaction on Software Engineering*, 1976.

[20] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *Transaction on Software Engineering*, SE-5(2), 1979.

[21] Parsytec Computer GmbH, Aachen, Germany. *PARIX Programmer's Manual*, 1991.

[22] J. L. Peterson and A. Silberschatz. *Operating System Concepts*. Addison-Wesley, second edition, 1985.

[23] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langois, P. Leonard, and W. Neuhauser. CHORUS Distributed Operating Systems. *Computing Systems Journal*, 1(4), 1988. Also as Technical Report CS/TR-88-7.9, Chorus systèmes, Paris.

[24] W. Schröder-Preikschat. Overcoming the Startup Time Problem in Distributed Memory Architectures. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, volume 1, pages 551–559, Kauai, Hawaii, January 8–11, 1991. IEEE Computer Society Press.

[25] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.

[26] M. L. Scott, T. J. LeBlanc, and B. D. Marsh. Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 255–262, August 1988.

[27] P. Wegner. Classification in Object-Oriented Systems. *SIGPLAN Notices*, 21(10):173–182, 1986.

[28] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[29] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, volume 21 of *Operating Systems Review*, pages 63–76, Austin, TX, 1987.

[30] R. Zajcew, P. Roy, D. Black, C. Peak, P. Guedes, B. Kemp, J. LoVerso, M. Leibensperger, M. Barnett, F. Rabii, and D. Netterwala. An OSF/1 UNIX for Massively Parallel Multicomputers. In *Proceedings of the USENIX Winter Technical Conference*, pages 449–468, San Diego, CA, USA, January 25-29, 1993.