Object Orientation in a Family of Parallel Operating Systems *

Wolfgang Schröder-Preikschat
German National Research Center for Computer Science
GMD FIRST at the Technical University of Berlin
Rudower Chaussee 5, D-1199 Berlin, Germany

Abstract

Forthcoming massively parallel systems are distributed memory architectures. They consist of several hundreds to thousands of autonomous processing nodes interconnected by a high-speed network. A major challenge in operating system design for these architectures is to combine transparency with efficiency. The paper motivates the program family concept and object-orientation as implementation instrument to build parallel operating systems that can be suited to the individual needs of parallel applications and computer architectures.

1 Introduction

A major challenge in operating system design for massively parallel architectures is to design a structure that reduces system bootstrap time, avoids bottlenecks in serving system calls, promotes fault tolerance, is dynamically alterable, and application-oriented. These architectures are based on distributed memory, i.e., they are distributed systems. Parallel applications for these types of systems are distributed applications demanding system-wide message passing with very low latency and very high efficiency.

The communication performance problem in these systems is dominated by the message startup time. Basically, the message startup time determines how many processor cycles are lost to local application program processing by performing remote interprocess communication. The loss of "number crunching" power caused thereby, is not only due to the communication protocol overhead but also a question of the actual operating mode of the node [14]. A single-tasking mode of operation, e.g., does not imply any form of address space isolation. There is no need to protect either the tasks (since there is only one per node) or the kernel (since it only keeps track of the resources of a single task). Thus, the kernel is nothing but a communication library, sharing with the application task the same address space. Of course, this is inconceivable if a multi-tasking mode of operation must be supported on the node.

The choice of the proper operating mode depends on constraints defined by the application and the kernel architecture. There are a number of parallel applications that scale very well with the actual number of nodes. These applications call for single-tasking support on the nodes. The microkernel architecture [6] promotes the encapsulation of system services by user-mode tasks. This approach calls for multi-tasking support on the nodes even if only a single application task must be locally processed. In this case, the microkernel does not work for but against the parallel application if the (parallel) operating system was assigned to reduce the message startup time to an absolute minimum.

Only extremely lightweight operating system structures will overcome the startup time problem in distributed memory architectures – "*network bandwidth is rendered virtually insignificant*" [10]. Above all, the performance limiting factor is the per-node operating system software overhead and Taking into account state of the art hardware technologies, it is not the limited network bandwidth that signifanctly increases the message startup time.

State of the art parallel operating systems design, therefore, must obey the maxim not to punish an application by system functions which will never be used. This includes the entire spectrum of computer resources, ranging from memory space to processor cycles. An open, application-oriented operating system structure is required, with the application and not the operating system deciding which functions must be

*This work was supported by the Ministry of Research and Technology (BMFT) of the German Federal Government, grant no. ITR 9002 2.

0-8186-1060-3425/93 $03.00 © 1993 IEEE

supported and which must not. An approach must be followed in which an operating system is being understood as a *family of program modules* [13] and not as a monolith of more or less related components. In such a context, the parallel application is an integral part of a *family of parallel operating systems* and *object orientation* [17] then is the natural choice to design and develop such a family [4]. Thus, an application becomes the final system extension.

An example of the *family-oriented* approach is PEACE [15], the parallel operating system developed for SUPRENUM [7]. PEACE started in 1986 as an object-based operating system relying on the microkernel approach. Since that time it evolved to an object-oriented system in which the microkernel-based representation is only one variant of the operating system family. This metamorphosis is in progress since 1990.

Lessons learned from the SUPRENUM development led to the conclusion that a microkernel-based system organization is not lightweight enough for parallel architectures [14]. Similar observations have been made with porting both *Mach* onto a shared-memory parallel computer [2] and *Chorus* onto a high-performance RISC architecture [16]. Overcoming the performance bottleneck problem in distributed memory architectures forbids a single microkernel implementation aimed at supporting both a family of parallel operating systems and parallel applications. Rather, the operating system family must be extended by a *microkernel family* [5].

The paper presents the PEACE parallel operating system family. It demonstrates object orientation as the key for constructing featherweight, flexible, and high performance operating systems for massively parallel systems. The paper concentrates on the description of the family organization and on a detailed illustration of the cooperation between the various building blocks. A case study on the modeling of active objects in PEACE is presented to exemplify how the design of a family of parallel operating systems may benefit from an object-oriented implementation.

2 Family organization

PEACE is a framework for (distributed) parallel applications and provides a *Process Execution And Communication Environment* for distributed memory massively parallel architectures. Although specifically designed to support high performance parallel computing, the PEACE framework is also suitable for constructing (microkernel-based) distributed operating systems with realtime capabilities as well as object-oriented parallel computing platforms for workstation networks.

2.1 Functional decomposition

The global architecture assumes that a member of the PEACE parallel operating system family is constructed from three major building blocks. These building blocks are the *nucleus*, the *kernel*, and POSE (Fig. 1). 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 the distribution of the building blocks over the nodes of the parallel machine.

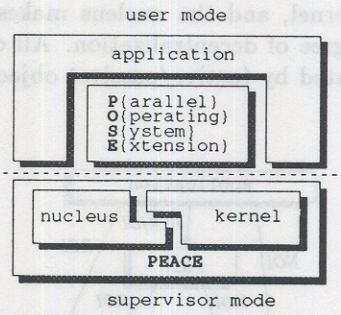

Figure 1: Building blocks

The nucleus implements system-wide interprocess communication and provides a runtime executive for the processing of threads. It is part of the kernel domain, with the kernel being a multi-threaded 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, interrupts). Application-oriented services such as naming, process and memory management, file handling, I/O, and load balancing are performed by POSE, the *Parallel Operating System Extension* of PEACE. [1]

Kernel and POSE services are built by active objects. In contrast, the nucleus is an ensemble of passive objects that schedule active objects. An active object is implemented by a *lightweight process*. A number

[1] In the following, PEACE is used as the synonym for both the framework to build parallel operating systems and the two fundamental building blocks nucleus and kernel.

of these objects may share the same address space, thus constituting a team of lightweight processes, i.e., a *heavyweight process*. Each service that is provided by both POSE and the kernel is implemented by such a team and represents a PEACE entity. Entities are system extensions. They are loaded on demand and (in most cases) can be arbitrarily distributed over the nodes of the parallel machine.

The dividing line between user and supervisor mode as shown in Fig. 1 is a logical boundary only. It depends on the concrete representation of the interactions specified by the *functional hierarchy* [9] (and of the hardware architecture) whether this boundary is physically present.

2.2 Invocation schemes

The functional hierarchy (Fig. 2) defined between POSE, the kernel, and the nucleus makes possible a very high degree of decentralization. All components are encapsulated by (active/passive) objects.

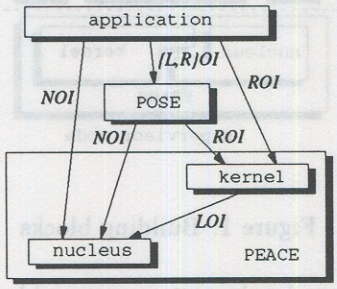

Figure 2: Functional hierarchy

Nucleus services are made available to the application via *nearby object invocation* (NOI). The logical design assumes a separation of the nucleus from the application (and POSE), which calls for the use of traps to invoke 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 with the entity the same node, but not necessarily the same address space segment.

The kernel resides with the nucleus in the same address space. Together they constitute the *kernel entity*. The kernel therefore performs *local object invocation* (LOI) to request nucleus services. Kernel services are made available via *remote object invocation* (ROI). The ROI scheme always implies context switching, but not necessarily address space switching. A separate thread of control is used to execute the requested method (i.e., service). In contrast to that, NOI implies the activation/deactivation of the nucleus address space via local system call traps. The implementation of ROI takes advantage of the network-wide message passing services provided by the nucleus and, thus, is based on NOI.

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

2.3 Actual structure

From the design point of view neither the kernel nor POSE need to be present on each node, but only the nucleus. In a specific configuration, the majority of the nodes of a massively parallel machine is 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 all not 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. The building blocks have been designed by considering the various schemes of object invocation (Fig. 2). However, it depends on the actual operating system family member whether these schemes become effective as specified by the design or can be replaced by a more simple and 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 *nucleus family* which place their focus on performance. The entrance to the nucleus is represented as an abstract data type with two implementations. The first implementation assumes no *vertical isolation* between PEACE (nucleus and kernel) and the using entities and no *horizontal isolation* between the entities itself. Thus, there is neither a separation between user and supervisor mode of operation (vertical isolation) nor a separation between competing tasks (horizontal isolation). In this case NOI actually means LOI. 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 two variants basically distinguish between single-tasking (no isolation) and multi-tasking (isolation) mode of operation. They are part of the nucleus family which in total consists of up to eight members [5], each one offering different performance characteristics and different functionality.

This organization of the kernel entity is one of the major differences between the PEACE approach and state of the art microkernels. The PEACE kernel entity is no single microkernel, but rather a "*microkernel family*". It can be adapted to the individual needs of parallel and distributed applications.

Offering dedicated single-tasking and multi-tasking kernel implementations enables the user to deal with the tradeoff between performance and functionality on an individual basis. For example, parallel applications whose tasks can be mapped in one-to-one correspondence with the nodes will never be punished by multi-tasking overhead. Experiences with PEACE have shown that this overhead consumes about 74 % of the message startup time [14]. For the SUPRENUM implementation, which is based on a 2 MIPS mc68020 processor, system-wide bi-directional interprocess communication using a 64 byte packet takes about 784 μsec in multi-tasking mode and 204 μsec in single-tasking mode. A state of the art microkernel however supports only multi-tasking [6] and, thus, unnecessarily drains computing power from a single-tasking application.

3 The role of object orientation

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 "program family" and "object orientation" (Fig. 3), it is almost natural to construct program families using an object-oriented framework [4]. Both approaches are in a certain sense dual to each other.

The minimal basis of system functions in the program family concept has its counterpart in the superclass of an object-oriented design. Minimal system extensions then are introduced by means of subclassing. Inheritance and polymorphism are the proper mechanism to allow that different implementations of the same interface (abstract data type) may coexist at the same time. 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" [13].

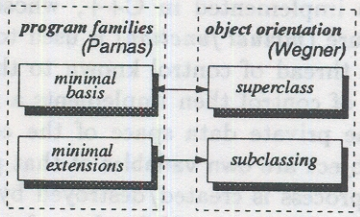

Figure 3: Program families vs. object orientation

PEACE has much in common with *Choices* [3]. Both systems share the same idea of an operating system family being object-oriented implemented. PEACE extends the *Choices* approach by the notion of a kernel family that offers to the user a number of operating modes for a node. Above all, the PEACE approach conforms to the program family concept. Object orientation is primarily used as implementation and not as design instrument. Inheritance makes it much easier to implement and maintain an incremental system design, but is not mandatory for it. The *design* (i.e., structuring) of PEACE was and is family-oriented, while the *implementation* was object-based and is now object-oriented.

4 Modeling active objects

As an example of how object orientation is used in PEACE and how all four building blocks of the entire architecture cooperate, the *process management subsystem* is discussed in the following. This subsystem implements active objects.

4.1 User view

An active object is constructed from a PEACE class that contains an *autonomous method* (`action()`) to specify the (private) thread of control of instances of that class. The method must be redefined by a specialized (application-oriented) class to enable the construction of an active object. Redefinition and thus specialization is accomplished by means of inheritance and crosses traditional protection boundaries. Thus, a parallel PEACE application is a PEACE specialization. The user is required to design a new subclass and to specify its own `action()`. One or several subclasses may be designed this way, to generate one or more process types.

PEACE is implemented in C++, whose dynamic binding feature (*virtual function*) is used to make the autonomous thread of control known to the nucleus. This thread of control then implements a *lightweight process*. The private data space of the corresponding active object are own variables of that process. A lightweight process is created/destroyed by executing the constructor/destructor of the class of an active object.

An active object acts either as **native** of the kernel or as an application, POSE, or kernel **thread**. For each case PEACE provides individual base classes whose constructors/destructors take care of the different ways to create/destroy the processes. Basing classes on **native** makes derived classes dependent on internal (nucleus and kernel) abstractions and requires that these objects be subjected to supervisor mode of execution. It serves to implement *site dependent entities* such as low-level device drivers. Basing classes on **thread** supports the implementation of *site independent entities*, whose executions are independent on the actual processing mode (user vs. supervisor) and on the effective destination node. Either way, the same **action()** specification is used to identify the entry point of the active object. This makes the association of a "user process" to either **thread** or **native** virtually transparent.

4.2 The thread

Almost every active PEACE object is a **thread** and, thus, is constructed from a class that is composed by *multiple inheritance* from POSE and nucleus classes (Fig. 4). [2] A **thread** instance is associated with a kernel-level active object, which effectively is implemented as a **native** thread (described in 4.3) – in fact, **native** scheduling yields **thread** processing. For the mastering of a **native** active object and, thus, of its associated **thread** instance, ROI to the kernel managing this object must be performed (arrow in Fig 4). For this purpose, an active object stores the address of a lightweight kernel process (*clerk*) used to receive, execute, and confirm the ROI requests. Mastering an active object means attaching it to a POSE entity (i.e., server) that performs process-related functions. For example, this is used to make name server known on a per-process basis and to define naming domains.

The construction of a **thread** instance results in the construction of an active object being managed by the kernel and, usually, in the creation of a (user) stack. A "kernel affair" is going to be established. In order to hide internals of the kernel (and the nucleus) from the clients, the general external interface on these active objects is specified by an *abstract class*. The interface class describes an abstract data type with different implementations, each one introducing a new process type. For threads being able to receive messages, each **thread** instance is bound to a system-wide unique identifier which serves as the ticket for system-wide interprocess communication carried out by the nucleus. Being in the possession of such an identifier (i.e., *sparse capability*) enables interprocess communication.

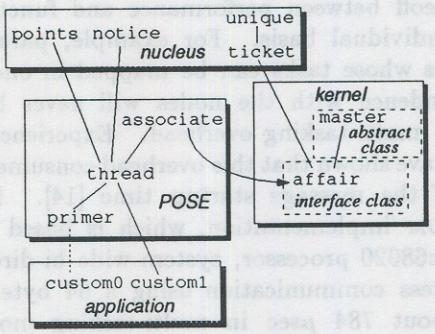

Figure 4: User abstractions

The kernel performs an *upcall* to invoke an active object when it has been initially scheduled, i.e., "bootstrapped". A specialized upcall handler then takes care of the invocation of **action()** in the **thread** context. Note, **thread** applies dynamic binding to "move the points" [3] for the upcall. Upon return from **action()**, the handler stops further processing of its enclosing **thread**. By performing NOI to the nucleus, it awaits a signal (i.e., notice) to invoke **action()** again and will never return to the kernel. Dynamic binding is also applied to specify the problem-oriented **action()** in the application context. The specialized upcall handler hence leads to the activation of a customized **action()**.

From the kernel (and nucleus) viewpoint a PEACE entity is activated as an ordinary task, which solely is represented by a **native** thread. At this point in time, only a single thread of control executes inside the entity. Further threads must be created by the entity itself. Upon startup, the local runtime system creates a **thread** root instance. The corresponding "primer" **action()** is nothing but the object-oriented

[2] Arrows mean "*uses*" [13] and lines additionally mean "*inherits*".

[3] *(Am.)* "shift the switch"

variant of the `main()` entry point of C++. [4]

4.3 The native

It was mentioned above that `native` scheduling yields `thread` processing. This corresponds to the traditional way of modeling processes in a multi-tasking environment which distinguishes between user and supervisor mode of operation. The process (logically) has two stacks, one for each operation mode. Depending on its execution state, the process is referred to as a *user process* (user mode execution) or as as *kernel process* (supervisor mode execution), with each mode executing on a different stack.

A `native` thread is a kernel process. It has an own stack only in the case of vertical isolation, i.e., when NOI is implemented as cross domain call. In a single-tasking environment without any isolation measures the `native` executes on the stack of its user `thread`. Thus, in these configurations the `thread` is processed in supervisor mode too. It shares with the nucleus (kernel) the same address space.

A *Process Control Block* (PCB) is used for each `native` thread to keep track of all process activities. It is a common approach in operating systems design that many different kernel modules share the same PCB for performance reasons. These modules, e.g., perform dispatching, scheduling, synchronization, communication, and resource management. They all store their process-related state information into a single data structure. Concerning PCB members, the sharing often is disjunctive, i.e., the kernel modules usually access only their own state information. However, they still have global knowledge about all the information stored in the PCB and thus are interdependent – they "*use*" [13] each other, which leads to a very poor internal system organization. The UNIX `proc` structure is a typical example of this PCB representation. Moreover, note that it is only the physical representation as a *flat data object* which yields performance – and not necessarily the logical one.

Inheritance is used to design the PEACE PCB and to introduce new process-related state information. Each class in the hierarchy is an *abstract data type* and, thus, allows state manipulation only via (*inline*) methods.

[4] Note that this procedure also supports non object-oriented PEACE applications, such as those ones written in C. The mentioned local runtime system contains a default `main()` implementation which invokes `peace()` to create the root instance. A user still may use its own `main()`, with the effect that no root instance is being created by default. In this situation only kernel-level process abstractions are used to implement a user-level thread. Later, the user may decide to enter the "object-oriented world" by simply calling `peace()`.

Since PEACE is implemented in C++, instances of (hierarchically composed) classes have a *statical representation*. The consequence is a flat data object representation of a single PCB and at the same time a highly modular kernel organization.

The class hierarchy (Fig. 5) to construct a PCB also serves a second purpose. It transforms a non object-based physical processor into an *object-based abstract processor*. At the lowest level (i.e., root of the inheritance tree) a thread of control is ordinarily represented by a program counter and other CPU registers. At a higher level the process becomes an active object encapsulating a private thread of control.

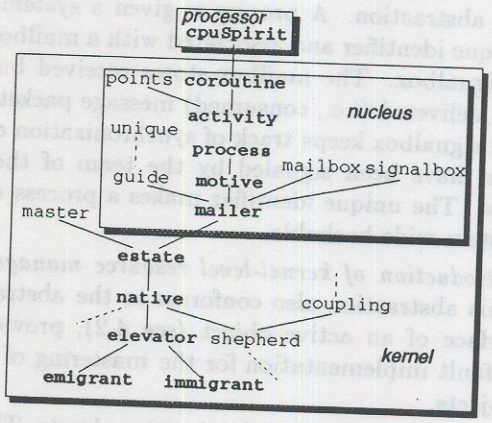

Figure 5: The process control block

The entire class hierarchy consists of ten levels (boldfaced symbols from top to bottom in Fig. 5, with the top being the lowest level), each of which implementing a *virtual machine* [9]. The corresponding class descriptions specify both the state (i.e., PCB fragment) and the operations of a virtual machine. The meaning of each level is as follows:

0 *Abstraction from the physical processor.* It describes the *non-volatile register set* as specified by the C++ compiler and used to save/restore the CPU context upon process switches.

1 *Introduction of concurrency.* Coroutines are used as abstractions from a single-processor machine and found the basis for low-level process switching.

2 *Abstraction from non object-based processors.* The autonomous method `action()` is declared as *pure virtual function.*

3 *Abstraction from the process dispatcher.* The dispatcher is *adaptive* and provides problem-oriented context switching capabilities. Context switching

may require saving and restoring other processor registers than the non-volatile register set, for example, the register set of a floating point unit used by "number crunching" processes. Dynamic binding is used at this level to reflect the various requirements.

4 *Abstraction from the process scheduler.* The scheduler is *adaptive* and provides problem-oriented scheduling capabilities. Dynamic binding is used at this level to introduce a per-process scheduler, providing a platform for a multi-scheduler environment.

5 *Introduction of message-driven scheduling.* This abstraction implements a refinement of the scheduling abstraction. A process is given a system-wide unique identifier and associated with a mailbox and a signalbox. The mailbox stores received but not yet delivered (i.e., consumed) message packets and the signalbox keeps track of synchronization events that have been signaled by the team of the process. The unique identifier makes a process object system-wide hashable.

6 *Introduction of kernel-level resource management.* This abstraction also conforms to the abstract interface of an active object (see 4.2), providing a default implementation for the mastering of active objects.

7 *Introduction of kernel-level active objects.* This abstraction serves as the general basis for dynamically created processes of any type. A *shepherd process*, e.g., is a thread of the kernel entity which is responsible for the handling of a specific interrupt. It is a coupling between a physical interrupt vector entry and a unique thread of control.

8 *Upcall linkage.* This abstraction stores the reference to an object used to upcall (i.e., bootstrap) a `thread` instance.

9 *Upcall handling.* There are two abstractions at this level, distinguishing between user and supervisor mode threads. For a user mode thread, additional processor registers are used to implement vertical isolation. For example, in order to transfer control from supervisor to user mode and, thus, let a `native` "emigrate" from the kernel to startup processing as a user-mode `thread`, the cross domain call must be furnished with the user-mode software prototypes of both stack pointer and program counter.

The design decision to distinguish between user and supervisor mode is met at a very high level of the kernel design. Active objects always start processing in supervisor mode and may switch over to user mode afterwards. Objects never leaving supervisor mode execution are considered as kernel immigrants. Objects entering user-mode processing (i.e., kernel emigrants) be subjected to vertical isolation and a *cross domain upcall* must be performed to transfer control accordingly. In either case, the objects is given a user-level visibility, making the PCB representation transparent even for kernel threads. This design decision is one reason for the fact that almost the complete PEACE (i.e., kernel and nucleus) implementation also runs as *guest level* on top of a host operating system such as UNIX.

4.4 Liaison of thread and native

In order to instantiate an active object, the kernel proper to create this object must be selected. A design decision was to address the kernel entity on a per-object basis, i.e., virtually give every active object its own managing kernel entity. Thus, the kernel plays the role of an abstract processor and any number of these processors may be used to build up an abstract parallel machine.

4.4.1 Dual objects

Kernel selection in a distributed environment means identifying the entity which manages the construction of some process instance. In PEACE, the constructor for an active object has a formal parameter that is used to address the managing entity of that object. A default address is provided for the ease of transparency. This default address selects the kernel entity that shares the same node with the client entity requesting object construction. The *service access point* (i.e., the system-wide unique identifier) of the kernel entity is stored with an active object descriptor in the client context. It is used for further operations on that object (and for its destruction) to identify the proper managing entity. Thus, the destructor of an active object will automatically find the way to the kernel entity that was responsible for construction.

This type of a remotely managed object is called dual object [12]. It has a dual representation, distinguishing between the client and the server site. Accessing the client site representation (*likeness*) automatically yields a ROI to the dual server site representation (*prototype*). Inheritance is supported in a distributed environment, crossing protection boundaries. Thus, providing *access transparency* on objects or object fragments.

For each representation of a dual object, a class specification is used. Both specifications are automatically generated from a *dual class* by applying the PEACE C++ class generator tool. The dual class is the original C++ class enriched by annotations (embraced in C-like comments) to describe the behavior of objects of that class. Basically, a generated client class is a subset of the generated server class. They both contain the public classes of the original class hierarchy specified by the dual class and they are extended by additional classes to support hashing and server selection. The server class furthermore contains all the private classes of the original class tree. Moreover, a *clerk* class is generated for the server site. This class is a **thread** and describes active objects that are capable of managing dual objects of the dual class from which the clerk description was generated.

4.4.2 Kernel linkage

The association between a **thread** and a **native** is physically implemented by a pointer to a dual object (broken arrow in Fig. 6).

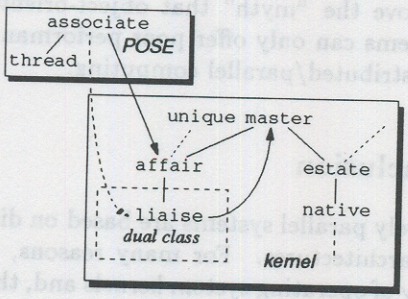

Figure 6: Liaison of thread and native

The likeness instance of this object consists of the system-wide unique identifier of the **native** and an interface object for **native** mastering, both considered as the "heirloom" for the dual class. The prototype instance contains the same information plus a **native** pointer (uparrow in Fig. 6). A kernel clerk receives ROI requests issued by the clients being in the possession of (and having applied) the likeness part. This remote residing lightweight server process performs for the clients the requested operation on the prototype instance and dereferences the **native** pointer. For example, the kernel clerk maps all operations for **native** mastering performed on the likeness onto the corresponding operations of the linked **native** object. The clerk thread is a kernel representative and its system-wide unique identifier is nothing but the address of the kernel entity managing process objects.

In order to make derived classes independent from using LOI or ROI to interact with the kernel, a **thread** only sees the "heirloom" for a dual class (solid downarrow in Fig. 6) and not the dual class itself. Thus, ROI to the kernel to construct/destruct and access an active object, is transparent to the user – it could even be a LOI for certain kernel configurations (see 2.3).

5 Discussion

A system that comes very close to PEACE is *Choices* [3]. Many ideas found in *Choices* are present in PEACE, and vice versa. This is because both systems share the same fundamental idea of a family of operating systems [9]. They extend this idea into object-oriented, distributed/parallel environments.

Like *Choices*, PEACE is a class-hierarchical system in which inheritance is excessively used to customize basic operating system abstractions. About 118 classes presently constitute the PEACE kernel entity, which corresponds to less than 10000 lines of C++ code. The minimal basis of system functions in *Choices* implements a multi-tasking mode of operation only. In contrast, PEACE further distinguishes between a number of operation modes and allows different implementations of the same abstract interface of the minimal basis coexist. The single-tasking kernel family member, e.g., optimally supports a single-tasking parallel application even in a distributed environment. Multi-tasking applications are supported by a corresponding multi-tasking kernel.

It is exactly this feature which becomes more and more important for forthcoming parallel operating systems. Without this kind of variety, a performance bottleneck problem is predefined in the system design. There are a number of parallel applications whose tasks can be mapped in one-to-one correspondence with the nodes of a parallel machine. However, they clearly will be handicapped if a multi-tasking kernel is the only choice to support parallel processing.

The PEACE approach goes beyond state of the art microkernel architectures [6]. It supports a scalable kernel architecture, in which a microkernel is only one concrete representation [5]. A microkernel is a fairly complex component, used to support the implementation of operating system services and the processing of distributed applications. The design assumes that even local system services are executed on top of the microkernel. This calls for multi-tasking mode of op-

eration on a single node and requires complete kernel isolation.

A microkernel is of course capable of executing single-tasking applications. However, it will only provide the adequate system support for multi-tasking applications if performance becomes the predominant issue. There is absolutely no reason to hamper "well-shaped" single-tasking applications in their attempt to get the utmost highest performance out of a parallel machine. Because of its scalable kernel architecture, PEACE bridges the gap between parallel systems and distributed systems.

Parallel computing defines its own "book of rules", even if parallel computer architectures are distributed systems. No matter which principle is used for designing a parallel operating system, the resulting implementation must never hide performance. For example, with the scalar performance of a 40 MIPS processor (RISC technology) in mind, a process-level and network-wide message startup must be performed in the order of magnitude of 10 μsec. In this case, a balanced ratio between the per-node computing power and network communication latency is achieved [11].

Porting distributed operating systems onto distributed memory computer architectures can be straightforward. However, it still needs a large effort to make these architectures work fast for applications which have little in common with distributed programming. Bespoke tailoring then means to think about what can be removed from the system design. Removing a function from a complete system which is not family-oriented organized and implemented is almost impossible. Even if being unexploited, these functions can be sources of (communication) performance loss because of their interdependencies with other system functions – they are not called (i.e., needed) but "*used*" [13] by the application. A multi-tasking kernel running a single-tasking application program is the typical example for this situation.

Choices, *Chorus*, and *Mach*, e.g., are faced with this problem. The former is family-oriented designed and relies on microkernel-like functions. The latter two are not family-oriented but microkernel-based designed. In contrast to these systems, PEACE is family-oriented designed and does not rely on microkernel-like functions. Compared to PEACE, their design does not support scaling down kernel functionality to an absolute minimum to meet the hard performance requirements of many parallel applications. Basing parallel applications on a microkernel platform requires that "*faster hosts are needed*" [10] to speed up performance. However, even if faster hosts are used, the performance bottleneck still remains the same. PEACE overcomes this bottleneck by its nucleus family.

PEACE is an object-oriented operating system family that corresponds to the classification that object orientation, above all, means inheritance [17]. Inheritance also means specialization. An object-oriented operating system must therefore use inheritance to compose internal system abstractions. It must further offer polymorphism for user customization. Thus, inheritance must cross traditional protection boundaries, such as to physically isolate supervisor mode entities from user mode entities. The PEACE design and implementation follows these maxims.

Using C++ to implement an object-oriented operating system family has many advantages – although the language itself is not in all cases the best choice. Because of the static nature of C++, almost no "inheritance tax" must be paid for having built class hierarchies. Since dynamic binding is not a default case, "tax return" is possible. By supporting *class-based object orientation* [1], most of the methods are represented as *inline* functions. This way, "tax abatement" can be asserted. All these aspects enable the implementation of *featherweight system components* and disprove the "myth" that object-oriented operating systems can only offer poor performance in the area of distributed/parallel computing.

6 Conclusion

Massively parallel systems are based on distributed memory architectures. For many reasons, they call for variety of operating system kernels and, thus, motivate the program family concept to design and develop system software support. Program families imply object orientation at least in the implementation process. This aids the realization of flexible, application-oriented, and high-performance parallel operating systems suited to the needs of massively parallel systems.

The PEACE approach presented in the paper is to define a framework to build a family of (parallel) operating systems. It goes beyond state of the art microkernel approaches and applies the program family concept also to designing and implementing a microkernel. A PEACE operating system thus is not supported by a single microkernel, but rather by a microkernel family.

PEACE is currently running as guest level implementation under SunOS 4.1. Ports onto a i860-based parallel computer and a Transputer system are almost completed. Future work encompasses the integration of fine grain load balancing, virtually shared

memory with strong and weak consistency, and persistency. Originally developed as a testbed, the guest level implementation will be extended to support parallel computing on workstation networks. The goal is to provide a single platform concept for a number of distributed memory architectures ranging from local area network systems to dedicated parallel computers.

7 Acknowledgements

I wish to express my gratitude to all members of the PEACE project. Especially to Jörg Cordsen and to Jörg Nolte for fruitful discussions on object-oriented operating system design.

References

[1] G. S. Blair, J. J. Gallagher, J. Malik, "Genericity vs Inheritance vs Delegation vs Conformance vs...", *Journal of Object-Oriented Programming*, Vol. 2, No. 3, pp. 11-17, 1989

[2] R. Bryant, Hung-Yang Chang, B. Rosenburg, "Experience Developing the RP3 Operating System", *Computing Systems, The Journal of USENIX Association*, Vol. 4, No. 3, pp. 183-216, 1991

[3] R. Campbell, G. Johnston, V. Russo, "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)", *ACM Operating Systems Review*, 21, 3, pp. 9-17, 1987

[4] J. Cordsen, W. Schröder-Preikschat, "Object-Oriented Operating Systems Design and the Revival of Program Families", *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, IEEE 91TH0392-1, pp. 24-28, Palo Alto, CA, October 17-18, 1991

[5] J. Cordsen, W. Schröder-Preikschat, "Towards a Scalable Kernel Architecture", *Proceedings of the OpenForum '92 Technical Conference*, Utrecht, The Netherlands, November 25-27, 1992

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

[7] W. K. Giloi, "The SUPRENUM Architecture", *Proceedings of CONPAR 88*, Manchester, UK, Cambridge University Press, pp. 10-17, 1989

[8] W. K. Giloi, W. Schröder-Preikschat, "Very High-Speed Communication in Large MIMD Supercomputers", *Proceedings of the International Conference on Supercomputing (ICS '89)*, Crete, Greece, June 5-9, 1989

[9] A. N. Habermann, L. Flon, L. Cooprider, "Modularization and Hierarchy in a Family of Operating Systems", *Comm. ACM*, Vol. 19, No. 5, pp. 266-272, 1976

[10] K. A. Lantz, W. I. Nowicki, M. M. Theimer, "An Empirical Study of Distributed Application Performance," *Technical Report STAN-CS-86-1117 (also available as CSL-85-287)*, Department of Computer Science, Stanford University, 1985

[11] H. Mierendorff, "Bounds on the Startup Time for the GENESIS Node", *ESPRIT Project No. 2447 Technical Report*, GMD F2.G1, Bonn, Germany, 1989

[12] J. Nolte, W. Schröder-Preikschat, "Modeling Replication and Placement in the PEACE Parallel Operating System – A Case for Dual Objects", *Proceedings of the Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems (ECOOP '92)*, Utrecht, The Netherlands, June 29, 1992

[13] D. L. Parnas, "Designing Software for Ease of Extension and Contraction", *IEEE Transaction on Software Engineering*, Vol. SE-5, No 2, 1979

[14] W. Schröder-Preikschat, "Overcoming the Startup Time Problem in Distributed Memory Architectures", *Proceedings of the 24th Hawaii International Conference on System Sciences*, Vol. 1, pp. 551-559, 1991

[15] W. Schröder-Preikschat (Ed.), "PEACE – The Evolution of a Parallel Operating System", *Arbeitspapiere der GMD*, No. 646, Germany, Berlin, 1992

[16] J. Walpole, J. Inouye, R. Konuru, "Modularity and Interfaces in Micro-Kernel Design and Implementation: A Case Study of Chorus on the HP PA-RISC", *Technical Report CS/TR-92-49*, Chorus systèmes, Paris, 1992

[17] P. Wegner, "Classification in Object-Oriented Systems", *SIGPLAN Notices*, 21, 10, pp. 173-182, 1986