Object-Oriented Operating Systems Design and the Revival of Program Families*

J. Cordsen W. Schröder-Preikschat

German National Research Center for Computer Science
GMD FIRST at the Technical University of Berlin
Hardenbergplatz 2, 1000 Berlin 12, FRG

Abstract

The 'myth' is disproved that object-oriented operating systems offer pure performance in the area of distributed/parallel computing. Rather, it is true that the object-oriented paradigm is the only chance to build high-performance systems while maintaining a clean and evolutionary system structure.

1 Introduction

Since 1986 the PEACE group at GMD FIRST is involved in the design and development of operating system software for massively parallel systems being based on distributed memory architectures. Massively parallel means several hundreds to thousands of processing nodes being interconnected by a very high-speed network and communicating via message passing. Primary goal was (and is) to design system software by consequently following the idea of program families [12].

The first PEACE prototype was a microkernel-based operating system family. Lessons learned from this implementation led to the conclusion, that the microkernel approach is not the ultima ratio to build operating systems for massively parallel distributed memory machines — and for others too. Furthermore we were forced to the conclusion that an operating system family based on generic libraries and description utilities for system generation isn't the overall approach to meet both distributed and parallel computing.

Although the PEACE communication performance is excellent for distributed systems, it is not for massively parallel systems. Problem is a significant portion of multi-tasking overhead (about 60–70 %) with each remote message transaction [15]. This overhead is implied by the microkernel approach, which at least requires to process a number of system tasks on each node. Most of the applications, however, are of single-tasking nature, i.e., they are distributed applications with a single task being mapped onto each processing node of the parallel machine.

Based on this experience a complete PEACE redesign was made. The microkernel approach was completely abandoned and a shift was made from object-based to object-oriented system organization. The new PEACE, also refered to as **objective** PEACE, is an object-oriented computing surface providing object support to user applications. An extremely small and efficient *minimal basis of system functions* builds a process execution and communication environment for massively parallel applications. By means of *minimal system extensions* custom-made microkernels and, thus, operating systems can be constructed.

2 Approaching the Concept of Program Families

In the program family concept a *minimal subset of system functions* provides a common platform of fundamental abstractions. This minimal basis encapsulates solely *mechanisms* from which more enhanced system functions can be derived. It will be built by a consequent postponement of design decisions. Fundamental abstractions to make massively parallel systems work then are *processes* and *communication*, i.e., message passing.

Dependent on the individual application, a stepwise functional enrichment of the minimal basis is performed by means of *minimal system extensions* only. These extensions encapsulate *mechanisms* and/or *strategies*. However, it might be the case that no sys-

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

tem extensions are necessary at all. The application itself is always the best extension one can think of — it is the final extension any way. A minimal basis which supports threading and communication already suffices to execute parallel programs. Moreover, it could be considered as the only operating system support residing on a processing node and being required by the application.

By adding minimal extensions, an operating system family is constructed bottom-up, whereby construction is controlled top-down: lower-level components are introduced only when required by higher-level components. This way, system functions for scheduling, security, process management, (virtual) memory management, exception handling, file handling, checkpointing and recovery are introduced. An open, application-oriented and evolutionary system organization is the consequence.

Understanding functional enrichment as an add-to in terms of components is only one aspect. It also includes *component replacement*. During the design phase, a commitment on a minimal subset of system functions must be made, with the risk of stating wrong design decisions. The processing of parallel applications by a massively parallel machine always implies communication, hence the need for communication functions. The application might also call for a single or multi-threaded address space (i.e., task) on a node. Another application demands multi-tasking, which then is a functional enrichment of multi-threading. Should the minimal basis therefore support multi-tasking ? If the design decision advocates multi-tasked nodes and tasks are mapped in one-to-one correspondence with the nodes, then a significant degradation of the message startup time will be the result [15]. Multi-tasking is not free of charge, even if not utilized by the application. The design decision to support solely multi-tasked nodes would have been met too early.

Nevertheless, common to all applications is the same external interface of the minimal basis. What differs is the internal behavior, i.e., the implementation. The external interface is mainly concerned with communication, while the internal behavior mainly dictates the process model and the operation mode of the node. With the minimal basis being an *abstract data type* [10], a number of implementations of the same interface can coexist. This makes the minimal basis exchangeable at least from the design point of view. Flexibility is maintained although the minimal subset of system functions must have been fixed early in the software design process.

Applying these design principles of program families lets the microkernel appear as a number of custom-made extensions to a minimal basis. It is merely a member of the operating system family: the bridge to distributed systems is being built.

3 The Object Paradigm and Common Operating Systems

It is quite known and scientific proved for a couple of years that the object paradigm offers an extensive support in the construction of complex software programs. Nevertheless, object-oriented programming still fights for acceptance in the area of distributed/parallel operating systems. A few research groups, i.e. projects that are free from commercial commitments, meanwhile started new object-oriented implementations from scratch.

The following is an attempt to state the object paradigm in well known operating systems, ignoring support layers and programming environments for distributed systems. An amalgamation with these layers may be appropriate for distributed systems, but not for massively parallel systems.

Despite of the vagueness of the term *object-oriented operating system* the following categorization is made with respect to Wegner's definition [18] and based on pertinent literature.

The designers of **Amoeba** [17] call their system object-based. This classification holds with system servers being active objects. Each server acts as an abstract data type, encapsulating state information and providing a set of operations available through remote procedure calls. Thus the number of objects is fixed through the amount of servers. The Amoeba design philosophy leads to a more considerable object world than operating systems which act as a distributed UNIX clone.

The fortune of **Chorus** [13] is struck by a two-edged sword. Originally designed and implemented by Inria/France, later adopted and commercialized by Chorus systèmes. While Chorus systèmes consequently pursues requirements of commercial systems — the latest Chorus V3 supports binary instead of source compatibility for Unix — Inria is concerned with object-oriented support layers for distributed applications on top of existing operating systems. This leads to a lot of experience with the object-oriented paradigm and at the same time forbids a reflection in its own implementation.

At the moment, **Mach** [1] is the most favorite distributed operating system, thereby characterizing state-of-the-art techniques and the indecision in use of object-oriented programming. While the designers boast about their Mach 3.0 microkernel, the distribution guide announces that the multi-server environment will take a long time to go, since it is built from scratch using object-oriented programming. This is in contrast to the actual microkernel implementation (100.000 lines of C-code with about 350 goto-statements) and symbolizes an attempt to fit out a house's attic on top of wobbly foundations. Nevertheless, according to [5] Mach 3.0 was designed to support object-oriented programming based on an object-oriented kernel interface. However, revising a former interface and refrain classes and inheritance will at best offer an object-based interface.

Although previously stated as object-oriented [16] the **x-kernel** [9] is merely object-based. The kernel is written in C, and the protocol and session objects just provide an uniform set of operations thus fulfilling an advanced object-based topic.

Examining **Clouds** [4] lets the object paradigm come more into light. The kernel implementation (12.000 lines of C++ and 1.000 lines assembly code) is object-oriented, but system objects are restricted to kernel space. User level objects are confronted with a system call interface, which is motivated by efficiency reasons, thereby introducing an invincible gap to object-orientation. However, Clouds is an object-based operating system based on an object-oriented kernel.

With respect to the systems mentioned so far, **Choices** [2] is the only object-oriented operating system. Representing the idea of a family of operating systems through a class hierarchy and inheritance offers a flexible mechanism to derive adequate operating system support. The basic Choices abstraction, however, only provide multiprocessing, thereby being to restrictive in a massively parallel computing environment.

4 The PEACE Approach

PEACE uses a class hierarchy and inheritance as the fundamental paradigm to build both a message passing kernel family and an operating system family, with the former being an integral part of the latter. The *minimal basis* of the class hierarchy is constituted by objects for single-processing and synchronous communication. *Minimal extensions* then are abstractions for threading, address space isolation, multi-tasking and multi-user/multi-tasking. Thus PEACE goes beyond Choices and embodies not only the notion of customized operating systems but also of kernel, that are tailored for particular hardware and software configurations. Currently the nucleus of the message passing family encompasses 89 classes.

Designing PEACE as an object-oriented operating system family from the hardware layer up to the application level interface enables not only to fulfill the requirements of distributed programming but also to satisfy specific demands from massively parallel computing environments [11]. These demands call for a systemwide message startup time in the order of magnitude of 10 μsec (40 MIPS processor).

At the root of the family tree, support of single-processing is made feasible with a kernel that takes the form of a communication library which is directly linked to the application task. This introduces the notion of *featherweight processes* [8]. These processes are not involved in any security measures. They can be compared with coroutines being subject to extremely lightweight thread scheduling. Since application and kernel share a common address space, context switches are reduced to the simple task of saving/restoring registers. The leafs of the family tree are multiprocessing kernels supporting standard multi-tasking and/or multi-user multi-tasking functions with address spaces being isolated.

The minimal basis of objective PEACE is always built by *passive objects*. These objects are instances of either class *message* (systemwide packet-based synchronous communication), class *transfer* (systemwide data transport between peer address spaces) or class *landmark* (systemwide unique addresses). Minimal extensions, however, can be *passive* as well as *active objects*. The extensions can be understood as being horizontal, vertical or both. In the horizontal case a complete new abstraction, i.e class, is introduced. A typical example would be the addition of threading. Vertical extension is by means of component replacement and specialization, e.g to introduce protection boundaries, dedicated network drivers and problem-oriented protocol machines. An example for the combination of horizontal and vertical extension is dynamic management of process objects. A new process subclass is introduced, by derivation from a subclass describing kernel processes, and made available to further derive customized (user) processes. This way, the kernel-level process control block is built by a hierachy of eight classes to make the non-volatile CPU register set a process (i.e, active object) executing in its own address space.

The extension by means of an active object would be necessary to enable remote object construction/destruction. These active objects (*clerks*) are made hidden and handle remote (in the sense of disjoint address spaces) object invocation requests. Whether the specialized family member ends up in a pure passive or passive/active representation is application dependent. If no one-to-one correspondence between application processes and processors is given, the resulting PEACE member consists of both passive and active objects. Otherwise, passive objects are the building blocks.

5 Upcoming Work

Presently (July 1991) our vote for an object-oriented PEACE operating system family is based on

- experiences from cooperations with research groups for parallel computing on a 320 node supercomputer [6] using object-based PEACE,
- commercial and research projects on distributed systems using object-based PEACE,
- knowledge from a guest level implementation of objective PEACE being the minimal extension for threading (8.000 lines of C++ and 100 lines assembly code),
- preliminary results from an ongoing port of the guest level implementation on a parallel machine consisting of eight nodes each equipped with two Intel i860 processors.

In addition, design and implementation of object-oriented language level support for remote object invocation was finished. It is in use to automatically generate *clerks*, i.e active PEACE objects, as required.

Dynamic (re)configuration receives great attention in daily work, since it is stated for years that configuration is a core issue to meet the needs for massively parallel systems. In the PEACE approach an overall object invocation mechanism implies that the kernel itself can be subject to dynamic reconfiguration [14]: e.g. replacing a single-processing kernel by a multiprocessing kernel.

Nearly all research groups that focus on parallel/distributed operating systems spend large efforts in the area of *distributed shared memory* or *virtually shared memory*, as we call it. In connection with object-based PEACE we were engaged to embed the shared memory paradigm in a distributed memory architecture [7]. Although this still is an important and essential topic, the challenge of objective PEACE motivated new ideas. We are now concerned with a form of object mobility, in which the protocol for maintaining consistency is an inherent object feature.

The application is confronted with a class hierarchy of consistency protocols. Specializing abstract class definitions through dedicated implementations provides an environment by which the required consistency maintenance is made feasible. A framework for the specification of consistency protocols is a simple and elegant way to adopt application-oriented strategies. Not least, the demand for such a framework is backed by mechanisms like *external pagers* in Mach. A detailed description of these *vagrant objects*, called *hobos*, is in progress [3].

References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, M. Young, "Mach: A New Kernel Foundation for UNIX Development," In *Proceedings of the Summer 1986* USENIX *Conference*, pp. 93–112, July 1986.

[2] R. Campbell, G. Johnston, V. Russo, "Choices (Class Hierachical Open Interface for Custom Embedded Systems)," ACM *Operating Systems Review*, Vol. 21, No. 3, pp. 9–17, July 1987.

[3] J. Cordsen, "VOTE for PEACE," *In preparation.*

[4] P. Dasgupta, R. J. LeBlanc, M. Ahamad, U. Ramachandran, "The Clouds Distributed Operating System," To appear in IEEE *Computer*, August, 1991.

[5] R. Draves, "A Revised IPC Interface," In *Proceedings of the Mach Workshop*, Burlington, Vermont, USENIX Association, pp. 101–121, October 4–5, 1990.

[6] W. K. Giloi, "SUPRENUM: A Trendsetter in Modern Supercomputer Development," In *Proceedings of the 2nd International* SUPRENUM *Colloquium*, Bonn, Germany, Sep. 30–Oct. 2, published in Parallel Computing, Vol. 7, No. 3, pp. 283–296, 1988.

[7] W. K. Giloi, C. Hastedt, F. Schön, W. Schröder-Preikschat, "A Distributed Implementation of Shared Virtual Memory with Strong and Weak Coherence," In *Proceedings of the 2nd European Distributed Memory Computing Conference*, Munich, Germany, pp. 22–31, April, 1991.

[8] W. K. Giloi, W. Schröder-Preikschat, "Programming Models for Massively Parallel Systems," *International Symposium on New Information Processing Technologies '91*, Tokyo, Japan, 1991.

[9] N. C. Hutchinson, L. L. Peterson, "The x-Kernel: An Architecture for Implementing Network Protocols," IEEE *Transactions on Software Engineering*, Vol. SE-17, No. 1, pp. 64–76, January, 1991.

[10] B. H. Liskov, S. Zilles, "Programming with Abstract Data Types," SIGPLAN NOTICES, Vol. 9, No. 4, 1974.

[11] H. Mierendorff, "Bounds of the Startup Time for the GENESIS Node," ESPRIT *Project No. 2447*, German National Research Center for Computer Science, Institute GMD-F2.G1, 1989.

[12] D. L. Parnas, "Designing Software for Ease of Extension and Contraction," IEEE *Transactions on Software Engineering*, Vol. SE-5, No. 2, pp. 128–138, March 1979.

[13] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, W. Neuhauser, "CHORUS Distributed Operating Systems," *Computing - Systems Journal*, Vol. 1, No. 4, pp. 305–370, Fall, 1988, The USENIX Association. Also as Technical Report CS/TR-88-7, Chorus systèmes, Paris, 1988.

[14] H. Schmidt, "Making PEACE a Dynamic Alterable System," In *Proceedings of the 2nd European Distributed Memory Computing Conference*, Munich, Germany, pp. 422–431, April, 1991.

[15] W. Schröder-Preikschat, "Overcoming the Startup Time Problem in Distributed Memory Architectures," In *Proceedings of the 24th Annual Hawaii International Conference on System Sciences*, Kauai, Hawaii, USA, Vol. I, pp. 551–559, January 1991.

[16] M. Shapiro, "Object-Support Operating Systems," Position Paper for the *Workshop on Operating Systems and Object Orientation*, July, 1990.

[17] A. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, G. van Rossum, "Experiences with the Amoeba Distributed Operating System," *Communications of the* ACM, Vol. 33, No. 12, pp. 46–63, December, 1990.

[18] P. Wegner, "Dimensions of Object-Based Language Design," *Special Issue of* SIGPLAN *Notices*, Vol. 22, No. 12, pp. 88–97, 1987.