Meeting the Application in User Space

H. Assenmacher[†], *T. Breitbach, P. Buhler,*
V. Hübsch, H. Peine, R. Schwarz

University of Kaiserslautern, Department of Computer Science
Erwin-Schrödinger-Straße, D - 67663 Kaiserslautern, Germany

In modern programming environments, the increasing variety of programming models cannot be satisfactorily matched by a single operating system interface. An interface adaptable to specific application needs requires a user-level realization of system services. The operating system kernel should be reduced to minimal functionality, not biased towards a particular programming paradigm. To provide flexibility in the construction of interface families, we propose a system architecture based on a small set of orthogonal abstractions.

1. INTRODUCTION

Today's operating systems still offer a static interface, not designed for customization by the application programmer. At the same time, application requirements are getting more and more specific, demanding higher flexibility from the underlying system. Emerging needs are, in particular, adequate support for concurrency, distribution, and persistence. However, there seems to be no commonly accepted single interface satisfying every kind of user application.

Rather than trying to meet diverse user requirements by further broadening existing interfaces, a more suitable approach is to supply highly customizable operating system services [5]. These services should be accessible in user-space like any application-specific service in the chosen programming model. For example, in an object-oriented programming model, the programmer should be able to apply inheritance to existing classes of operating system objects. Such flexible access to existing primitives enables each application to configure its environment according to specific needs, yielding an interface *family* in contrast to a single ponderous interface.

The PANDA project [2] is an attempt to address these issues. The goal of PANDA is to provide an environment for parallel and distributed programming in C++, imposing as few as possible restrictions on the use of the language. PANDA offers class hierarchies handling parallelism, distribution, and persistence. Besides directly using these classes as a ready-made runtime environment, they may also be extended and modified to create more specialized programming systems. By supporting the coexistence and cooperation of several runtime environments providing different operating system interfaces, multi-model programming may be introduced [4, 8].

2. SYSTEM ARCHITECTURE

The PANDA architecture consists of a very small operating system kernel, and several alternative runtime packages located in user space. Besides flexibility, the driving motivation behind user-level service implementations is to avoid the performance penalty of crossing the privilege border by

[†] Email: assen@informatik.uni-kl.de

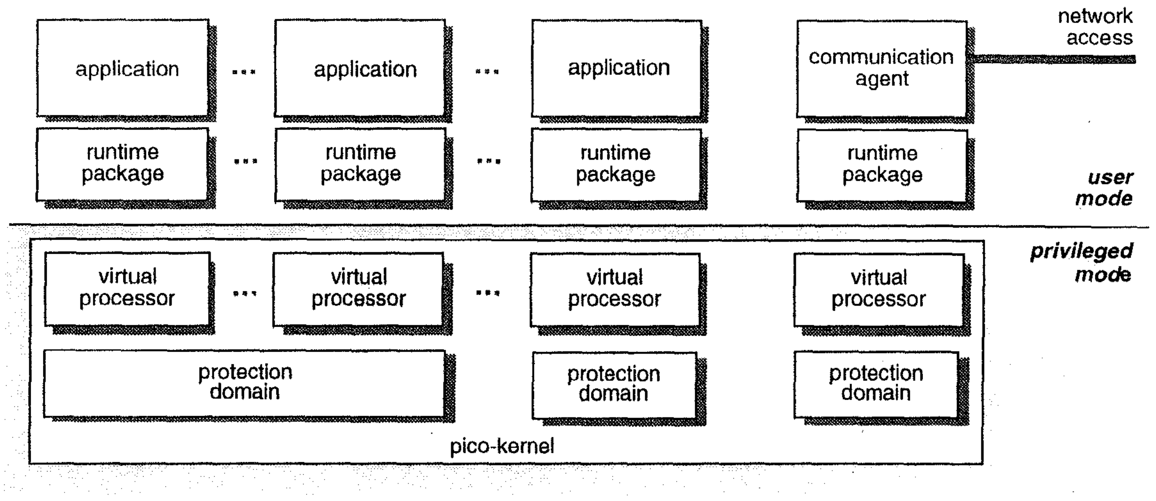

Fig. 1. The PANDA system architecture

reducing the kernel *call frequency*. Conventional kernels, on the other hand, move lots of functionality into the kernel to reduce the relative kernel *call overhead* compared to execution time spent on each call. In the context of parallel, object-oriented programming the applicability of the latter approach is limited. Note that self-contained objects ensure their integrity using internal locking. This entails intensive use of synchronization calls whose relative overhead can hardly be reduced. By moving such functions into user space, substantial performance gains may be achieved.

Therefore, the PANDA kernel provides only those functions that are relevant with respect to protection and monopolization. All other services are realized in non-privileged execution mode. The operating system kernel is called *pico kernel* to emphasize its underlying principle of minimality [3]. From our point of view, such an architecture is a prerequisite for the degree of flexibility necessary to realize an interface family.

The kernel offers the two abstractions *protection domain* and *virtual processor* (Figure 1). A protection domain controls the access to address spaces on a per-page basis. A virtual processor deals with interrupts, exceptions, and communicates with other virtual processors on the same node. At least one protection domain is assigned to each virtual processor. A single domain may be shared by several processors.

The major fraction of the operating system services are realized in a runtime package, potentially tailored to specific application needs. In order to simplify user-level thread scheduling, there is a one-to-one mapping between virtual processors and runtime packages. A substantial performance gain is achieved because there is no need to synchronize scheduling activities [1]. If an application wants to exploit the processing power of a multiprocessor platform, several RTPs sharing the same protection domain must be employed on each node, one for each physical processor. Since each RTP operates on its private system objects, frequent cache invalidation caused by system activities such as thread scheduling is avoided. Maintaining only a loose coupling between the processing units of a multiprocessor is a key to exploit its full computational power [7]. In order to achieve multiprocessor scheduling, threads have to be migrated between RTPs, either explicitly or by a load balancer.

One design objective of the PANDA architecture is to handle shared resources, devices in particular, outside of the kernel with dedicated service agents whenever possible. For instance, this

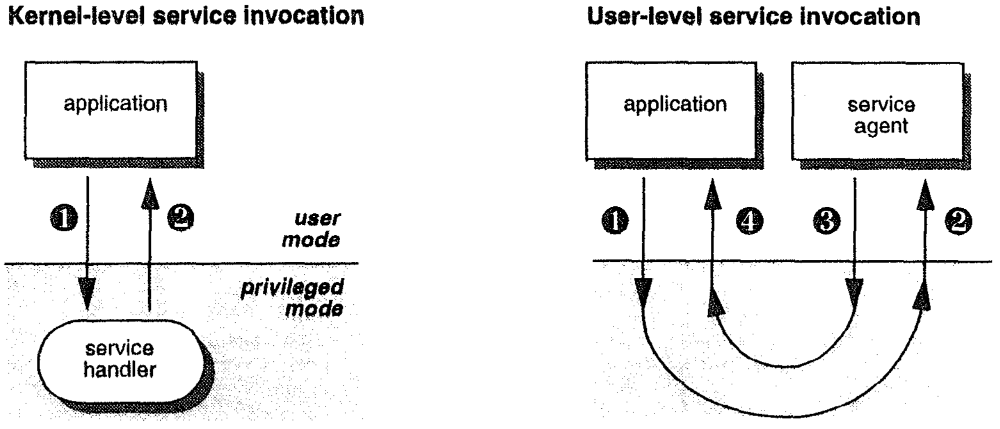

Fig. 2. Kernel-level vs. user-level service invocation

applies to PANDA's remote communication agent (see Figure 1) which is responsible for network access. By separating services from the kernel, better modularity is achieved, and thus extending and maintaining the system is facilitated. Moreover, service implementation can profit from PANDA functionality. Although such an approach is commonly accepted as desirable, its feasibility has often been questioned. One concern is whether privileged instructions are dispensable for device handling on a given hardware platform. In this respect, we did not face major problems with target architectures such as, for example, a 486 PC or a Motorola 68030 development board. Another concern is the potential performance loss induced by user-level service invocation. In a straightforward implementation, passing control to a service agent requires one additional context switch at kernel level compared to a service implementation within the kernel (Figure 2). This is generally regarded as a substantial penalty. However, providing support for efficient cross-domain switching has now been recognized as an important requirement for future hardware architectures. But even contemporary hardware yields reasonable performance. Currently, our (non-optimized) pico kernel implementation requires 83 µs on a 486/33MHz processor to perform steps 1 - 4 depicted in Figure 2. By application of the techniques described in [6], substantial acceleration is feasible.

In a shared memory multiprocessor workstation, a promising approach might be to reserve a dedicated processor preferably for service agents. In this case, agent invocation is reduced to simple signalling without explicit transfer of control. Thus, additional context switching is avoided.

3. ORTHOGONALITY OF MECHANISMS

PANDA's runtime package provides typical operating system abstractions as an integral part of the application. Therefore, similar design and development techniques apply for both the RTP and the user program, including the use of the same programming language. This enables seamless transitions between application and operating system services without any impedance mismatch. In the context of object oriented development, the flat interface layer of conventional operating systems transforms into a class hierarchy. The application is constituted simply as an extension of this hierarchy.

In order to obtain maximum flexibility, the RTP interface family has been based on a small set of fundamental, mutually independent abstractions. Among the basic conceptual building blocks are activity, a notion of mobility, time, and synchronization (Figure 3). They can be combined to realize the familiar abstract services. For instance, *time* and *synchronization* yield a delay operation; *activity*

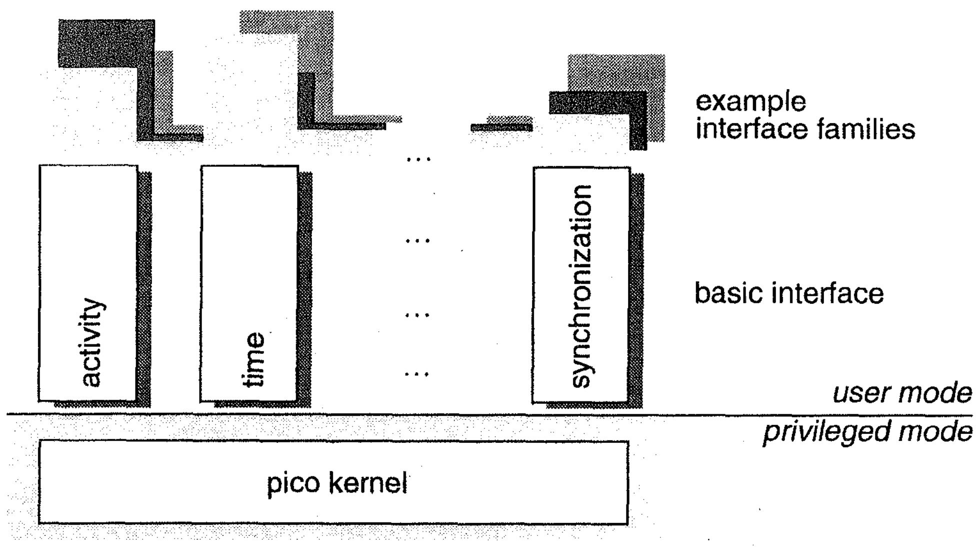

Fig. 3. PANDA interface hierarchy

together with *mobility* constitute thread migration, while *mobility* applied to passive objects realizes a global object space. The main challenge in the design of an interface family is to implement the basic concepts in an independent, orthogonal fashion so as to avoid early commitments and to allow interoperability. Mobility, for example, should provide mechanisms to "move an object around" regardless of whether it is an activity or (potentially persistent) passive data.

The relation between activity and synchronization may illustrate the importance of an orthogonal design. At first glance, it may appear natural to combine these concepts by equipping the activity class with synchronization primitives. For fast mutual access, even incorporating auxiliary synchronization data within activity data might seem reasonable. However, intertwining the implementations of synchronization and activity is liable to fail in the long term because it restricts RTP designers in their freedom to provide new synchronization mechanisms. If activity has to provide additional support for advanced synchronization policies, then all activity classes must comply with these extensions. In particular, any attribute added to synchronization — such as, for example, global accessibility of locks — must then be reflected by all application-specific derivatives of activity in use. Preferably, activity and synchronization should be kept apart. The former may simply be modelled as an entity which can be activated and suspended. The latter should then be a mechanism which provides an atomic flag with associated pending activities — without having any specific knowledge about the nature of activity.

Disciplined separation of high-level abstractions of an RTP is equally important. Services such as distributed virtual shared memory (DSM) or garbage collection which are autonomous in principle but may interfere with each other must be designed and implemented independently. Achieving orthogonality only at the *interface* level is clearly not sufficient. Mechanisms may turn out to be interdependent in such a way that their combination induces awkward restrictions for the application. As an example, consider the realization of sharing and activity by means of DSM and user-level threads. If DSM handling is realized as an ordinary thread, then — in order to avoid cascaded page faults — the thread scheduling mechanism itself must never cause DSM misses.

When introducing a new basic abstraction, the occurrence of non-functional interdependencies inevitably requires considering, and possibly revising all other existing abstractions. In the worst case, this may lead to an exponential increase in programming effort and, in addition, it is prone to error. To avoid unexpected interference, orthogonality at the language level should be adequately reflected at the architectural level. At least for activity, object and thread mobility, and persistence, we were able to achieve complete separation of concerns.

4. BUILDING DIVERGING INTERFACES

The PANDA project originated from diverging application requirements. On the one hand, we aimed at facilitating the transition from traditional programs written in C++ to distributed software. On the other hand, an experimental platform for the coexistence of active, migratable, and persistent objects was investigated. Both were realized as runtime packages representing different interfaces. They diverge in their model of activity, synchronization, distribution, persistence, and object granule.

The "traditional" RTP aims at supporting distribution with the highest possible degree of location transparency. A key issue in our design was to tolerate applications which are implemented in a rather impure object model, following the C spirit of C++. Typically, such applications heavily rely on pointer references. An adequate means to retain the expressiveness and efficiency of pointers in a distributed environment is a distributed virtual shared memory (DSM) spanning the local address spaces of all system nodes. DSM implementation is based on hardware surveillance and thus avoids penalizing local accesses. Our DSM protocol provides multiple read copies, and it guarantees sequential consistency. In addition to DSM, the RTP offers explicit or implicit, page-fault triggered, thread migration. Thus, a choice between two models of sharing — data or function shipping — may dynamically be taken. Global sharing implies appropriate synchronization facilities matching the application's access patterns. For example, one has to ensure that acquiring a read-only lock does not induce hidden write operations, thus invalidating the read copies of the corresponding memory page.

Our second, "explorative" RTP puts more emphasis on applications strictly following the principles of object-oriented design. All basic abstractions — among others activity, mobility, and persistence — are directly associated with language level objects. These fine-grained application objects constitute the unit supported by the underlying system. In consequence, an object granularity gap between general operating system support and application needs is avoided beforehand. This prevents phenomena such as false sharing and false locking. Unfortunately, contemporary standard hardware does not offer adequate object access surveillance. To this end, software solutions are mandatory.

Object sharing is based on establishing object co-location and thus avoiding non-local object access. A *tying* mechanism ensures symmetric and transitive propagation of co-location requests. In the case of temporary hardware dependencies, objects may explicitly be fixed to a given node. Due to orthogonal design, thread migration is simply subsumed under the tying of activities. Objects may as well be instantiated as persistent entities, surviving the life-span of an application or even a system crash. Persistent objects are typically — but not necessarily — accessed using the transactional guarantees of atomicity, durability, and isolation. Again, these objects may, for instance, be mobile. A distributed commitment protocol ensures consistency.

Our experience gained during the development and use of the outlined runtime packages demonstrates the benefits of the interface family approach. We were able to support the simultaneous presence of two models which considerably differ in most aspects of object invocation, mobility, synchronization, and persistence. This has been achieved by extending PANDA's basic interface without any modifications of the existing class hierarchy.

5. CONCLUSIONS

Apparently, it is not possible to construct a *single* system which provides an optimal set of mobility, activity, or persistence functionality matching the demands of any application. From our point of view, the need for an early decision in favor of a specialized operating system may be eliminated by supporting the coexistence of different interfaces. The flexibility required for diverging interfaces can only be achieved by moving operating system functionality into user space. The system is accessed through a user-definable interface family, adaptable to specific application requirements. In order to exploit flexibility to its full extent, it is essential to ensure orthogonality of basic mechanisms, and to bridge the granularity gap between system and application. This can be achieved with a pico kernel approach without undue performance penalties.

REFERENCES

[1] T.E. Anderson, B.N. Bershad, E.D. Lazowska, H.M. Levy. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.* ACM Transactions on Computer Systems, Vol. 10, No. 1, pp. 53-79, Feb. 1992.

[2] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, and R. Schwarz. *PANDA — Supporting Distributed Programming in C++.* In: O. Nierstrasz (ed), Proc. of the 7th European Conference on Object-Oriented Programming (ECOOP'93), LNCS 707, Springer 1993, pp. 361-383.

[3] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, and R. Schwarz. *The PANDA System Architecture — A Pico-Kernel Approach.* Proc. 4th Workshop on Future Trends of Distributed Computing Systems, Lisbon, Portugal, pp. 470-476, Sep. 1993.

[4] B.N. Bershad, E.D. Lazowska, H.M. Levy, and D. Wagner. *An Open Environment for Building Parallel Programming Systems.* Proc. of the ACM SIPLAN PPEALS - Parallel Programming: Experience with Applications, Languages, and Systems, pp. 1-9, July 1988.

[5] G. Kiczales, J. Lamping, C. Maeda, D. Keppel, and D. McNamee. *The Need for Customizable Operating Systems.* Proc. 4th IEEE Workshop on Workstation Operating Systems, Napa, CA, pp. 165-169, Oct. 1993.

[6] J. Liedtke. *Improving IPC by Kernel Design*. Proc. 14th ACM Symposium on Operating System Principles, pp. 175-188, Dec. 1993.

[7] J. Misra. *Loosely-Coupled Processes.* In: E.H.L. Aarts, J. van Leeuwen, and M. Rem (eds.), Proc. Parallel Architectures and Languages Europe (PARLE'91), Springer-Verlag, LNCS 506, pp. 1-26, Eindhoven, The Netherlands, June 1991.

[8] M.L. Scott, T.J. LeBlanc, and B.D. Marsh. *Multi-Model Parallel Programming in Psyche.* Proc. 2nd ACM Symposium on Principles and Practice of Parallel Programming, pp. 70-78, March 1990.