INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Multi-level Architecture of object-oriented Operating Systems

Sven Graupner, Winfried Kalfa, Frank Schubert

TR-94-056

November 1994

Abstract

Applications should be provided with optimal infrastructures at their run time. The proposed architecture encourages to structure a system into *sets of interacting instances* supported by optimal *infrastructures* at *multiple levels*. Infrastructures are organized as sets of instances as well, but of more elementary quality. Thus, a recursive architecture results with related infrastructures and instance areas that forms an n-ary tree. Each instance area provides the infrastructure for higher instance areas and needs itself a lower level infrastructure. Processing is considered as performing services among instances.

Object-orientation is proved to be suitable for structuring instance areas and infrastructures. Instances performing services are objects. A discussion of general principles of object-orientation gives the background to apply it to this architecture. Most existing object-oriented systems only consider one kind or "quality" of objects, which is however inadequate for operating systems. The paper discusses what essentially makes different "qualities of objects" at different levels and how activities are related to them.

In the last section the design and the implementation of a lowest level infrastructure is presented which is taken from an operating system prototype that follows the proposed architecture and which is under development in our group.

Multi-level Architecture of object-oriented Operating Systems

Sven Graupner, Winfried Kalfa, Frank Schubert

Chemnitz University of Technology, Germany
Department of Computer Science, Chair of Operating Systems,
TU Chemnitz, Fakultät für Informatik, 09107 Chemnitz, Germany
Tel.: (+49)-371-531-1390 / Fax.: (+49)-371-531-1530
email: {sgr,kal,fsc}@informatik.tu-chemnitz.de

November 29, 1994

1 Introduction

Operating system technology has now been investigating for over three decades. The separation of applications and the operating system was essential to cope with the complex control of machines, like multi-processing to exploit parallelism between the CPU and I/O-devices. Well understood structuring principles were developed, such as the concept of processes and their hierarchical order (*THE*, [Dijkstra68], [Dijkstra71]), the virtualization of system resources (virtual memory, *Multics*, [Bensoussan72]) for hiding machine resource limitations and providing idealized, abstract machines for applications (*VM*, [Creasy81]) to make them independently from proprietary hardware by providing standardized interfaces for common services (open systems, *UNIX*, [Ritchie74]).
Operating systems are complex, thus several structuring techniques were investigated and realized, such as layered structuring (*THE*), virtual machine structuring (*VM*), kernel structuring (*UNIX*), μ-kernel structuring (*MACH*, *CHORUS*), proxy structuring (*SOS*) and other. Is the subject of operating systems still to be investigated? Firstly, the theoretical concepts are hardly applied to "commercial systems" (most *UNIX* systems don't have process structuring in their kernels). Secondly, new requirements are permanently arising due to new features of the underlying hardware and by applications that want to make use of them. For this reasons operating systems remain a subject of research and improvement. Some points for new requirements are:

- the purpose of operating systems changes more and more from the pure managing of resources to providing optimal run time environments for applications with different needs,
- the integration of new media with the strong need of intelligent managing of insufficient system resources, such as scheduling with adaptable strategies or qualities of services,
- making higher performance available to applications by exploiting parallel processing (locally, remotely), the placing of processing to specialized machines with running dedicated systems, and thus location independent computation, storing and accessing of data,
- supporting long term distributed applications with permanent data (no global shutdown),

- the connection of machines by networks and applications making use of that, distributed processing needs distributed control over non-centralized resources, networks are not just connected computers, but form a society of resources that are shared among the users, computer networks develop more and more towards the use of personal communication,

- one special need which should not be ignored is keeping compatibility to older or other systems by providing their interfaces, but not necessarily promoting their structure,

- typical non-computer devices are becoming more and more complex and are realized by software which also needs some kind of operating system to run.

The upcoming of new features or requirements has always been present in operating systems. One way to deal with this is to add all new functionality to existing operating systems. This led to fat systems in the past. Another way is to realize new functionality outside the kernel. Typical μ-kernel systems follow this direction radically. The μ-kernel only provides the functionality for the existence and interaction of instances and basic device control. All other is performed by server-instances outside the μ-kernel. New servers may easily be added or replaced.
An example for the extension of an existing operating system is the networking facility of UNIX. The primary communication was built inside the kernel, such as the network drivers, the protocol stack with IP, TCP/UDP and the sockets interface for applications. An application, the Network File System (NFS) was added on top of UNIX. It is not part of the kernel, but is it therefore not part of the operating system? An operating system is not just the kernel, it encloses all instances providing *general or common services* for applications.

But, there is a structural problem. The traditional separation in just two parts: the kernel- and the application-level is inadequate. The NFS demon processes should *not* be considered at the same level as applications making (indirectly) use of them. Consequently, it is not reasonable to provide infrastructural functionality for instances by other instances at the same level.
For example, considering a high-level application using a CORBA-like infrastructure that is based on DCE that itself is placed on top of UNIX is complicated to be understood and to manage by having a mass of UNIX processes using the same infrastructure, the same scheduling policy, the same priority control, paging strategy, etc. . Instead, there should be dedicated infrastructures, which optimally suite to the needs of application instances running on them.
Multi-level structuring is needed and the relation between the area of instances and their infrastructure providing general services is relative, as shown in figure 1. Instances at one level provide (optimal) infrastructural services for instances at a higher level, which themselves provide the infrastructure for even higher instances. This leads to a *recursive or multi-level structure* of the system. This is more than the traditional layered structuring. An infrastructure could possibly provide services for multiple disjunct instance areas that form each dedicated infrastructures for higher instance areas. This structure forms in contrast to layers an n-ary tree.

This view is general enough and could be applied down to the hardware crossing different levels of infrastructures and instance ares. Using a PC it is possible to run different operating systems, each providing different infrastructures for applications, or virtually even both of them simultaneously, as NT provides it for MSDOS and UNIX. The model also applies to the hardware elements such as the CPU that needs an infrastructure to work (power, clock pulse, etc.). Although this is of less interrest in the context here, it shows the generality of this view.

The point of interest is, which levels and which infrastructures are suitable to structure operating systems, what are their properties affecting instance areas, and how they can efficiently be constructed on current hardware.

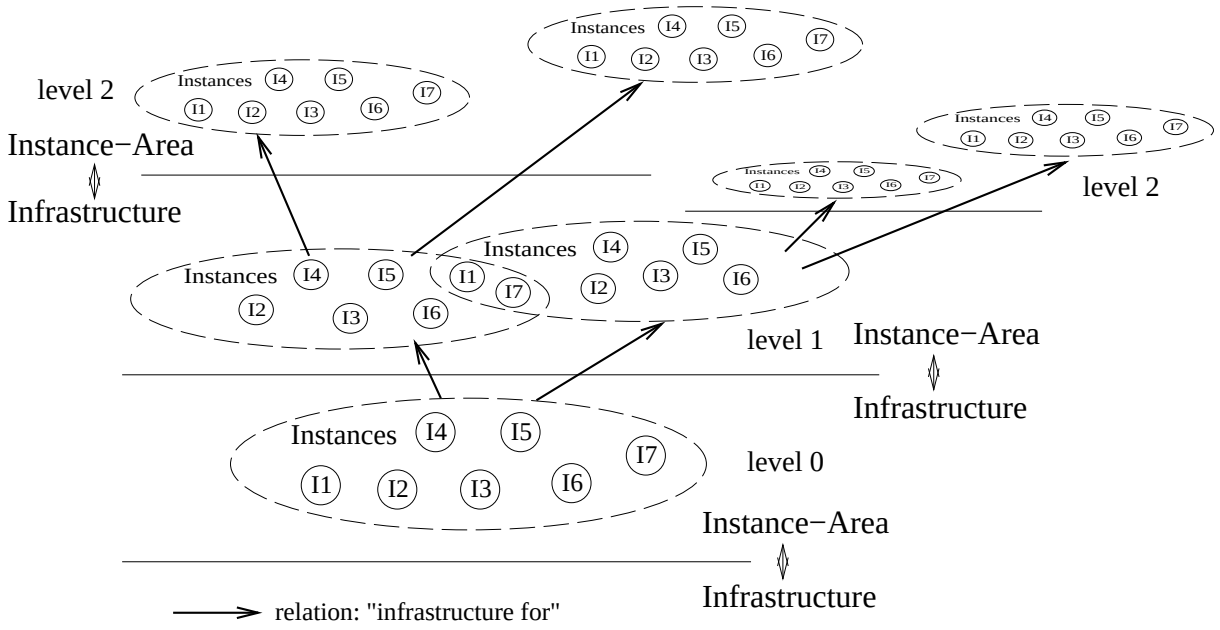

Figure 1: Multi-level Architecture with Instances and Infrastructures

Object-orientation is becoming more and more popular in structuring applications. There should be an infrastructure that optimally supports these applications at run time. If object-orientation is beneficially for structuring applications, it is obviously to apply it to structure the infrastructure too. Combining object-orientation with the introduced instance–infrastructure relationship leads to a homogeneous object-oriented structuring concept with multiple levels down to the hardware. Different levels mean different kinds of objects and relations which is detailed below. It is however necessary to examine general principles of object-orientation firstly that could be used for structuring a running system. This will be discussed in section 2.
The main points for new challenges in designing operating systems are summarized:

- multi-level system design with a relative relationship between *instances* and their dedicated, optimal *infrastructure*, which forms an *n-ary tree* (instances form the infrastructure(s) of higher-level instances),
- this *recursive* system structure breaks at a defined level 0 infrastructure, which is defined as the hardware infrastructure for operating systems,
- different levels are primarily distinguished by *general properties* of instances,
- the infrastructure should a priori consider the potential extensibility of a system by setting up new infrastructures as instance areas of a lower infrastructure, which provides the

necessary flexibility without beating performance because of the specialization of such infrastructures towards the needs of applications,

- *object-orientation* should be applied as a structuring method for instances and infrastructure across different levels.

2 Does Object-Orientation suite to Operating Systems ?

Object-orientation has been increasingly becoming popular in many fields of computer science over the last years. It is mainly applied to the different phases of the software development process as analyses, modeling, design and programming. The ideal goal is to provide a homogeneous structuring method, unified abstractions and views to a system at the various *descriptional* phases and subphases, to smooth transitions among them and to support all of this with tools. This is the origin and (still unsatisfied) idea of object orientation. Object-orientation came up in programming languages as the consequence of their improvement towards higher abstractions to reflect "real world things" closer in programs. This development was very well outlined in [Marty89].
Object-orientation is also becoming popular in structuring *running systems*, as databases and operating systems. Its is evident that this is another background for object-orientation as the production of software. An operating system is running on a machine providing an abstract machine to run applications. It is an active system, that primarily needs control, which is given by its own software, that must otherwise be produced some way. Not the production of software will be in the focus here, but really existing instances at run time like processes, their control and interaction to do some work.

2.1 Current object-oriented Approaches in Operating Systems

There are currently four general approaches for applying object-orientation to operating systems:

1. the *language approach:* object-orientation is only applied to the production of the code of an operating system by exploiting the features of an object-oriented programming language, there is no object-oriented structure at run time (monolithic kernel),
2. the *object management approach:* object-oriented applications are directly supported by a non–object-oriented infrastructure, languages are of less importance, there are two approaches:
   (a) a real operating system running on bare hardware,
   (b) run time extensions hosting on traditional operating systems and extending them,
3. the *client-server approach in μ-kernels:* server processes are instances providing services (= objects) and a μ-kernel forms the infrastructure, operating system functionality is provided by server processes,
4. the *multi-level approach:* the integration of application- and system-levels within a homogeneous, recursively structured architecture of multiple levels of instances and infrastructures in a running system.

Examples for these approaches are:

(1.) *Choices* [Russo91], which is implemented as a framework of C++-classes. It was developed at the University of Illinois. Applications use services of the system by creating instances of system classes (e.g. files) and accessing them by their methods. A proxy mechanism is used to cross the user/system barrier. This approach makes use of the object-oriented facilities of C++ at the program level. The running kernel is monolithic where objects are just "portions of data in memory", as it is defined for C++ in [Stroustrup90]. Activities or processes, which are essential for operating systems, are not covered by reducing object-orientation to the means of a sequential object-oriented language.
ETHOS [Szyperski92] is another example for this approach. ETHOS was developed at the ETH Zürich and is implemented in OBERON [Wirth88].
In [Müller91] a proposal is made to represent different layers of an operating system as instances of classes representing one layer each. All components of a layer are aggregates of this class. The interface of a layer can directly be reflected as the interface of a class. Orthogonally to the hierarchy of layers a hierarchy of specializations is applied to put the system to different hardware. However, there is just the language level considered.

(2.a) *BirliX* [Härtig90] was developed at the GMD, Bonn. It is a full operating system running on bare hardware and supports distributed applications that are structured as collections of "ADT-instances", or objects, which run each in a "team"-environment, which is a multithreaded process with its own address space. Instances communicate via RPC.

(2.b) The origin of the *SOS* system is the INRIA, France [Shapiro89]. It provides run time support for distributed objects (fragmented objects) in C++ concerning communication and location. The SOS features are integrated at the language level by extending C++ by additional keywords and using a translator to automatically produce the native code (proxies).
A similar direction has *PANDA* [Assenmacher93] of the University of Kaiserslautern by extending the run time environment of C++ to distribution, persistence, object migration and multithreading. The work of [Müller92] also extends the C++ run time system for distribution of objects across a network of heterogeneous machines.
An example for a language independent commercial system in this category is *DCE* of the OSF [Schill93] which provides run time and development support for distributed applications in a large-scale, heterogeneous network of workstations. All DCE services are realized by processes, mostly running on UNIX. Instances providing services are objects.
All these approaches provide extensions for insufficient application environments of traditional operating systems, they are layers or infrastructures built upon these systems.

(3.) The *MACH*-kernel [Accetta86] of CMU does only provide the (minimal) infrastructure for the existence and communication of server-processes that actually provide the operating system functionality. Such processes are instances providing services and can therefore be termed as objects. Some commercial systems are based on such sets of instances, as OSF/1 or NextStep.
The UNIX interface (or other) can fully be provided by sets of servers, as shown in [Golub90]. *CHORUS* [Rozier88] is very close to MACH. *NT* of Microsoft has a similar direction. The practical need for compatibility of interfaces to older or other systems is covered by providing their interfaces but with a different internal structure.
Own experiences were made in our group by restructuring *MINIX* [Tanenbaum90] in an object-oriented fashion. The new system was called *MINIX++* [Wiatrowski92] and also follows the μ-kernel approach.

(4.) Whereas other examples could easily be found for the above approaches, there are only a few systems covering this point. The most advanced operating system here is *Apertos* [Yokote92] of Sony CSL and Keio University, Tokio. The basic structuring principle is object–meta-object separation. All infrastructure for instances (objects) is provided by meta-objects forming a meta-space. A set of instances share the same meta-space. Other sets may have other meta-spaces. Infrastructural services are accessed by a so called "meta-of-link" to a "reflector" which is the visible proxy of a meta space for instances.
Other examples for this structural approach are *PM* [Kleinöder92] of the University of Erlangen and *PEACE* [Schröder93] of GMD. But both of them do not enforce multi-level structuring. In PM everything are objects at the same level. PEACE distinguishes specialized "families of operating systems" that are comparable to specialized infrastructures, but there is no multi-level structuring.

2.2 Generalized Concepts of Object-Orientation

Generally, *object-orientation is a method for structuring problem spheres by dividing them into objects and relations among them and to provide a structure to reduce the complexity.* But, the nature of a problem sphere heavily influences the kind of objects and their relations. This is mostly left aside when the discussion is about objects and object-orientation. The same (well known) terminology is applied to different problem spheres implying a total different meaning of the terms. In the pre-object-oriented age different things were named with different terms. There were types, variables and procedures at the program level, and processes running these programs at the system level. Today everything are objects. An analytical description is made of classes and objects, the design, the program and even the running system consists of objects that are naturally supported by system objects. Objects at run time are *not* the objects at the program level, as instances of C++ classes. In this case, they are just part of the code and the data of an active instance in the running system, such as a UNIX process. The example shows, that there are different problem spheres which are reflected here as different "worlds" of objects. Consequently, the background must always be clear if object-related terms are used. But there is also a lack of understanding the general concepts, principles and methods which are behind all that different spheres where object-orientation is applied today. The first part of this section will therefore concentrate on this topic. Some related discussions are in [Snyder93], [OMA92], and [OMG91].
Finding generalized concepts of object-orientation requires the analyzing of different approaches, beside the already known principles from programming languages. Three cycles will be presented here. At first, a most general approach will be discussed, which is later refined to different object qualities in computer science, and last, to objects in running systems, in particular to operating systems.

2.2.1 Viewing a World made of Objects

The high degree of abstraction the topic comes with forces to treat it appropriately. It is really possible to view everything existing in our surrounding as objects. The question: "What is an object?" can be seen closely related to the philosophical discussion of the "nature of things".

A general definition of the term "object" can be found in [Web88]:

> *object:*
>
> 1. a thing that can be seen or touched; material thing that occupies space
> 2. a person or thing to which action, thought, or feeling is directed
> 3. what is aimed at; purpose; end; goal
> 4. Gram.: a noun or other substantive that directly or indirectly receives the action of a verb, or one that is governed by a preposition
> 5. Phil.: anything that can be known or perceived by the mind ...

What can be learned from viewing the world around us as made of objects, and what does it mean for computer science? Primarily, things or objects do really exist in nature or in the universe, they're not just human reflections in mind. Things or objects are distinguishable and have properties, which constitute their existence, state, and behavior. Objects or things are consisting of other, more elementary objects or things of equal or more elementary qualities. There is an inherent recursion. The recursion is broken at a most elementary level, which are as far as we know the particles of atoms. There is permanent motion, change and transformation driven by energy. One phenomenon of the universe of things is the ability of self-organization, that is particles come together under certain conditions forming new things of new qualities, like an atom of a chemical element or a chemical compound.
What makes up a new quality? There must significantly new properties arise that only come up when certain elements are put together under certain circumstances. This provides a structure or order among the elements. In nature it is a permanent process of change and evolution controlled by its own. All the a priori existing things or objects around us are the product of this process, even the human being itself. The human being is an example of such a quality, it is not just an accumulation of carbon, water and other. It is a really existing individual. This view can be generalized. The same basic elements can form new things of new qualities and can be viewed at these different qualities. A new quality makes up a new shading of reality, not virtually, but really.
Things or objects are identifiable by having at least one criterion that distinguishes them. This depends on the quality where individuals are viewed. At the level of atoms, plants, animals and human beings are not to be identified, they all are just huge collections of them. Only the properties of a certain quality give them an own identity.
Things or objects are not unrelated to each other. There are relations among objects of the same quality and to constituting objects of less qualities. Relations among equally leveled objects are constituted by lower level objects. Viewing the world this way leads to some principles that also apply to object-oriented structuring in the concerned topic. They are summarized:

- things or objects do *really exist*, that is independently of human reflection, they are part of reality of a certain quality,
- a new *quality* requires *significantly new properties* that only arise at a certain combination of elementary objects under certain circumstances, the *organization* of lower quality objects brings up new qualities, that make new "kinds" of objects,
- there must be criterions by which individual things or objects may be distinguished and which *identify individuals relatively* to a certain *quality*,

- objects are formed by arranging more elementary objects of the same (*aggregation*) or less quality (*recursion*), the recursion breaks at a most elementary quality,
- the same part of reality may be viewed at different qualities (*granularity*),
- objects have *properties* that constitute the existence, state and behavior of objects, properties are *relatively to qualities*,
- *relations* among objects are based on relations of their constituting objects.

Interpretations could be found for these general points for any concrete object-oriented system. However, in most systems there is only one quality or kind of objects considered, which is referred to as the "object model" of the system. Promoting a multi-level structure of operating systems requires to distinguish different qualities of objects (section 2.4).

2.2.2 Models as intellectual Reflections of Objects

The next related topic is the recognition of reality by man. Recognition means the intellectual reflection of real world objects and their relations in mind. This is primarily based on the perception of sense organs for a limited spectrum of signals sent out by objects of the surrounding. Human intellect does not only notify such signals, but is able to detect patterns, rules or regularities to simplify recognition by means of abstraction. Abstraction is an intellectual process that results in an idea or *model* of a certain part of reality. Abstraction is always directed to a goal or purpose. Methods are to put a focus on a part of reality of a certain quality, to collect and order facts about it, to identify relevant individuals by finding criterions distinguishing them, to extract their relevant properties and relations inside and to the outside world, and group them to "higher" or more abstract entities in mind by finding similarities (classification). Such "higher" entities have an own semantical meaning, an own identity within the model and own properties. They need not necessarily exist in reality. "Higher" entities can be grouped themselves, which also leads to a recursive structure of the model.
Of course, abstraction leads to a loss of realism. Ideally, all phenomenon appearing in the viewed part of reality should fit into the model which was made of this reality (correctness).
The intellectual model reflecting a part of reality can be *described* by means of certain languages (mathematical formulas, graphical symbols, artificial or native languages) to make it available to other people that can understand them or to machines that can interpret them. Descriptions are reality themselves.

To summarize:

- recognition is always related to a certain *part of reality* and to a certain *quality* (focus),
- *abstraction* is an intellectual process to reduce the complexity of the viewed reality that results in a *model* in mind directed to a goal or purpose, *methods of abstraction* are:
  - the finding of criterions distinguishing individuals, their properties, and relations within the viewed world (*identification*, naming),
  - the *selection* of relevant individuals oriented to a goal, their relevant properties and relations, and leaving out other,

  - the ordering and grouping of semantically related or similar individuals, properties and relations on certain criterions (*classification*),
  - the intellectual construction of "higher" individuals out of such groups that have an own identity, properties and relations with an own semantical meaning within the model (*generalization*), which leads to a *recursive structure* of the model,

- models can be "materialized" by *describing* them by means of artificial or native languages.

2.2.3 The Production of new Objects

The discussion above concentrated on viewing a world of a priori existing objects with different qualities and their intellectual reflection in mind that leads to a model of the viewed world. But there are also "artificial objects" as the result of man's production. Production means the composition, transformation, or processing of raw material to produce new things of a possibly new quality. Production is directed to a goal or purpose, which is based on an idea or model in mind. A model is not only a reflection of the surrounding reality, a model can be intellectually changed, combined, or adapted to certain requirements or purposes. The goal of such a process could be to have a model of new objects with new properties or qualities that are to be produced. This model reflects a product specification, its later properties and use.
The model is the basis to derive a technology for production that later gives control to the production process. The technology includes to specify the raw material to be used, which tools are needed, where the production takes place, who does it, and gives a sequence of instructions to be carried out. There must be acting individuals carrying out the production. They get control by the instructions previously described in the technology.
All mentioned items must be provided by the infrastructure where production takes place. May be, that parts of the infrastructure must previously be produced themselves, such as needed tools. Here another facet of recursion comes up. This recursion must be broken somewhere. Producing "material" things fits well to this consideration, like pots, cars or houses. They are later real existing things or objects of a certain quality with an own identity and properties as discussed above. The "production of software" will be discussed in the next section. Some points to summarize are:

- production follows a goal, purpose or idea in mind that is the *model of the product*,
- from the model a *technology* is derived to specify all prerequisites for production:
  - the raw material needed, tools, acting individuals carrying out the production which is controlled by a sequence of instructions.
- the *infrastructur* must provide all prerequisites for production that possibly must be produced themselves before (*recursion*),
- the products are later part of reality with their own identity, properties and quality.

2.3 Objects in Computer Science

Computer science deals with the processing, transmission and storing of information upon a viewed reality by using machines which process input data to output data. For this purpose a representation of the viewed reality must exist inside the machine, that reflects the states and the behavior of the viewed reality but actually provides the control of the machine that interprets this representation to perform some work. The representation must previously be produced some way as well and is then reality itself. It exists in form of coded bit patterns in main memory in digital computers today. To produce the representation, a model of the viewed reality must exist. This way, there are four different "worlds" to be distinguished, all of them are in accordance with the statements in the sections 2.2.1, 2.2.2, and 2.2.3 above:

1. the viewed part of *reality* information should be processed upon,
2. the *model* reflecting this reality to some degree as a prerequisite for
3. the *production of a representation* inside the machine (manually and/or with tools) as a *description of state and behavior* for later *controlling* the
4. *processing* of input to output by the machine, which is the actual goal of that scenery.

The first point is not of interest here. The second is the subject of analysis, which is primarily an intellectual process to find a model (section 2.2.2) upon a part of reality. The model in mind is then materialized to descriptions, which are the basis for multiple transformations of descriptions (design, programming, compilation, linking, loading into main memory) until the description results in a memory representation that actually can be interpreted by the machine and gives control to it. The transformations of descriptions are necessary to close the gap between the modeled reality and the reality of the machine a representation is produced for. Processing is the goal, producing the representation a needed matter, that can be supported by tools to some degree.

Methods of object-orientation are primarily applied to the subject of the second and the third point. The big promise of object orientation is to provide a homogeneous structuring concept covering the analysis and the production of descriptions and all their transformations (design, programming). However, object-orientation mostly stops at the point where no more manually transformation is necessary. After the compiler has done its job the structure of objects is often not longer kept.

Object-orientation enforces the structuring of a certain "world" (which also implies a certain quality) into identifiable objects with identifiable properties. Properties constitute the state, the behavior and relations among the objects and to the outside. The purpose of object-orientation in computer science is to structure information processing by means of *performing services* requested and provided by objects.

Object-orientation reflects the abstraction process discussed in section 2.2.2. Properties are classified into *sets of common properties* and such sets are put into relation, which forms a *hierarchy*. The properties of one set result as the unification of all property sets at higher stages (*inheritance*–relation among property sets). Objects are related to one or more such sets that defines their properties.

Criterions for the classification of property sets are:

- common properties of *all* objects in the viewed world,
- properties of *equal* objects, with the same behavior and state set, but different current states and identities,
- properties of *similar* objects by summarizing equal properties and differentiating others,
- properties of *single* individual objects.

Relations among objects and properties are restricted to relations to classified property sets. In programming languages this is reflected by the object model (point 1) and the class hierarchy (points 2,3,4). [Wegner90] terms the classification of property sets of equal objects as "functional abstraction", the classification of property sets as "super abstraction". Properties are mostly represented by variables (states) and methods (behavior) there. In a running system there are additional things to be considered, such as activities and their interaction. Object-orientation encourages the *generalization and centralization of properties* reflecting the hierarchical abstractions of a model as much as possible, and thus also encourages the multiple use of property sets (reusability).

Encapsulation permits the view to a system as a set of objects with certain behavior (semantics) and interactions among them. For their use there are only those properties visible which are important for interaction. These properties constitute the *interface* of the object. All other properties are internal or *private* and hidden. That's the idea of modularization [Parnas72] (information hiding = property hiding). Providing encapsulation allows viewing a system at a higher, less complex level, that directly reflects abstraction as detailed in section 2.2.2.
In programming languages this is the concept of abstract data types (ADT) that (ideally) fully describes the behavior of instances of such types without any implementation details. In practice, there is mostly the syntax of the interface described and the description of the semantics must rely on appropriate names and comments. Formal approaches for describing semantics by sets of rules mostly fail on complexity. By the way, C++ classes are not truly ADT's because they also include the implementation additionally to the interface. Although, interface inheritance may be applied to simulate ADT's [Müller91].
Encapsulation is often distinguished from protection. But, if encapsulation really hides internal properties (e.g. states) then protection is guaranteed by encapsulation. If not, there is no real encapsulation, or it is enforced only at the language level, but not at run time, as in C++.

Identification should be separately treated from naming, addressing and locating of objects. Identification is always unequivocally both way, naming must not. The heavily discussed point of polymorphism is just the summing up of differently realized, but semantically related properties under the same name, which is related to the topics of encapsulation and abstraction above. Polymorphism refers to the naming of interface properties of objects, that have equal semantical meaning, but a different internal representation. However, it does not affect the identification inside the system, but simplifies the view of a user. Addressing relates to the place where an object currently resides (may change) and locating means the finding a path to it. The differentiation is of course not always necessary, which depends on the quality of the objects. But a general view should be pointed out here. This shows again, there are different qualities needed to cope with the variety of needs, not only in operating systems

Aggregation means the composition of objects of the same quality to form a more complex object of the same quality. It is essential that all involved objects are at the same level. Orthogonally to the hierarchy of property sets there is a second *hierarchy of aggregates*.

There are various *relations*, such as:

- relations among sets of properties in the hierarchy (*inherit_from* or *is_a*),
- relations among objects and one or multiple of such sets (*is_instance_of*),
- relations reflecting the aggregation hierarchy among composite objects (*is_part_of*), and
- relations among objects performing services (*uses* or *client-server*).

It should be stressed, that there are different interpretations for concrete worlds of objects. The above discussion concentrated on just one level, as it is mostly found in current object-oriented approaches. The goal however is to have different levels of instances and their infrastructures. The infrastructure for a world of objects must at least provide means for the:

- *existence,* that is the internal representation that determines the lifetime of an object from its creation until its destruction;
  before an object may be created, a description of its representation must be known to the infrastructure to enable it to create new objects, it is possible:
  - to make just instances of already present descriptions of representations, or
  - to provide the infrastructure with new such descriptions,
- *identification,* to make an object known to other objects for the purpose of interaction with several possible variations for naming, addressing, and locating,
- *interaction,* that objects form an active system to do some work, interaction should be possible:
  - among objects within the instance area, or
  - with objects within the infrastructure,

  interaction is closely related to *activities* and *control* which is discussed in detail in section 2.4.1 and 2.4.2 below.

Some of the objects of the infrastructure are identifiable or visible to the objects of the instance area. They provide infrastructural services for them. The set of such objects and their services forms the *explicit* interface of the infrastructure to the instance area. This is comparable to the interface of an operating system kernel. But infrastructure may not be reduced to services that are explicitly accessible by instances. Also the memory belongs to the infrastructure where instances are represented in. Of course, memory can be viewed as an infrastructural-object, but its servicing methods are not explicitly invoked by an instance. This is in contrast to services like opening a file, where accessible proxies are identifiable and referable for instances.
There are objects in the infrastructure that are not visible to the instance area. Their services however are essential for the existence and interaction of instances. Compared again with an

operating system kernel, a user process must have an representation inside the kernel (process control block). This representation is an infrastructural object that is not visible to the instance area. There is no interface and there are no services provided for instances. This can be viewed very close to the discussion of encapsulation as a general principle of object-orientation above. The infrastructure encapsulates all private properties to instances that are however essential for their existence and makes only some of them visible for them. These properties form the *infrastructural interface* which instances can explicitly use.

If objects of the instance area make use of infrastructural services that influence their own constitution this is called *reflection* following a similar mechanism which is applied to systems of artificial intelligence [Maes88], [Kiczales91].

The above points are a proposal to summarize the various facets of object-orientation in computer science. In the next section these points are applied to running systems, operating systems in particular.

2.4 Object-Orientation in Operating Systems

In contrast to object-oriented descriptions, operating systems are active systems controlling the processing of application instances and providing them with the required resources. Operating systems must also control their own processing, which makes them especially complicated and which is another reason for the insufficiency of a one-level approach. Objects cannot provide their own infrastructure fully by their own. Therefore at least two (or as proposed more) levels of infrastructures/instances must exist. The criterion for distinguishing levels are qualities with significantly new properties of infrastructures that are provided for different sets of instances. Qualities are different kinds of state sets of objects, different kinds of activities and interactions. A "quality" of interrupt handling activities may be structured by objects performing services, a "quality" of UNIX-like processes as well. But both should not be mixed up.
Object-orientation encourages the centralization of *general properties* of a set of instances into their infrastructure. There are different such properties. This is another reason for multi-level structuring to achieve optimal run time environments for applications.

Even in object-oriented operating systems the main concern remains the management of resources and the control of running applications (locally or distributed) and its own control. This is independently from any object-orientation. Dividing the complex processing into interacting objects provides a clearer structure. Processing is considered as performing services among objects. Objects themselves are constituted by services of lower level objects in the infrastructure. In contrast to most single threaded applications, operating systems contain a variety of activities of different qualities. It is incomplete to reduce object-orientation to the production of code by using a sequential object-oriented programming language. The active structure of an operating system cannot fully be grasped at this level, because the language has no means for expressing things like multiple activities, even not at different levels.
Although, there are many proposals for introducing concurrency to object-oriented languages by creating new concurrent languages as *ABCL/1* [Yonezawa86], by extending existing languages as *PRESTO* [Bershad88], and *μC++* [Buhr92], or just providing libraries with `fork()`-like routines, as *C-Threads* [Cooper87], *pThreads* [Graupner94] or other. In operating systems there are different kinds of activities, such as activities handling interrupts up to UNIX-like user processes, which should be separately treated.

The conclusion is not to build an universal language that covers all features, but to recognize the systems structure not just at the language level, instead to use the language just as a tool to produce code for system components and to assemble them later at the system configuration.

Special attention must be given to the relations among objects and activities, which is detailed in the sections 2.4.1 and 2.4.2. Section 2.4.3 explains different qualities of state sets, activities and their interaction as indicated by our experience.

2.4.1 The Nature of Activities and Control

Objects in operating systems are primarily identifiable entities providing services for requestors. The service itself is performed by an activity. In single threaded applications there is only one activity (e.g. C++), which makes it superfluously to especially deal with activities and which is the reason why object-oriented programming languages does not much concentrate on activities. There are mostly procedure calls for requesting services for efficiency.

An *activity* is the controlled doing of an active unit in a system. Control is given by a sequence of instructions the active unit must be provided with. A unit is active if it is able to get its control instructions itself and carries them out. One example of such a unit is the CPU. But there are also I/O-controllers doing something. The clock unit belongs to them. Its doing is just to decrement internal counters and signaling their zero states. Even the control units of the bus or the main memory are such units. Their control is hardwired. But from the point of effects there is no distinction between hardware or software control. Of course, the control of the bus or the main memory is fully hidden (encapsulated) in the infrastructure compared to the program level of the CPU.
The examples demonstrate the variety of active units inside a machine and their different levels. There is a hierarchy of control of different levels. Good system design makes sure that tasks of control of one level are hidden to higher levels, such as the bus control or main memory control to the CPU program level. Every computer has thus a priori multiple active units and multiple activities that are in parallel with each other. This parallelism is exploited for multiprocessing. The effects of activities are mostly sequences of state changes, at least to tip one bit somewhere or to do some I/O.

There are three general granularities of control which active units must have:

(1)-control: the coarse control of the unit, such as to stop, to interrupt or to continue it,

(2)-control: the sequence of instructions the unit has to carry out in each cycle,

(3)-control: the control that defines the processing of cycles itself,

(1)- and (3)-control are common for all instances and should therefore be part of the infrastructure. Each active unit has such three granularities of control in any form. Two examples should be discussed. A disk controller has (1)-control for putting it into action to do an I/O job, initiated by the CPU. How to handle this job is controlled by the (2)-control, which is either given by previously downloaded code (if it is programmable) or it is hardwired control. Of course, the single actions must also be controlled themselves some way, which is the (3)-control. For a CPU these three controls are: (2)-control is given by the program the CPU carries out, (3)-control is the job of the internal CPU-control, (1)-control is for the coordination with the bus-control.

At the level of the CPU there is only one flow or thread of (2)-control defined by the instructions read from memory. The model of von Neumann machines exactly includes this. But real machines have many other active units as shown and coordination is needed among them. Some coordination is hidden to the (2)-control of the CPU, such as the bus access, other not. How does the CPU running a stream of instructions become aware of coordination conditions? There are two general principles:

- fully (2)-control, there must previously special instructions be included in the stream of instructions to test data for a coordination condition and to branch if it is true (data-oriented or polling), or
- the (1)-control in the infrastructure has the ability to become aware of such conditions and automatically inserts another stream of instructions in the actually processed one (control-oriented or interrupt).

The first principle does not require any ability of the infrastructure, but it needs previously to take any combination of possibly multiple coordinations into account. It is hard to handle such programs. The second principle releases programs from that burden, but needs an infrastructure with the interrupt (1)-control facility, as found in any CPU. It is interesting to mention that inside the CPU interrupt cycles are detected by polling state bits after each instruction cycle (polling), which is an example for different views to the same thing at different levels.

There are some well known structuring principles for (2)-control, such as branches that only affect the location of the next instruction in memory. Other are procedures or coroutines with additionally keeping own processing state. All of them have in common, that the control is explicitly transferred to such units (next instruction, procedure or coroutine). Explicitly means it is described in the program.

But, the unpredictable insertion of instruction sequences, that is not previously considered at (2)-control, may be insidious because of critical sections that may occur by manipulating common resources between the interrupted and the inserted sequence. This violates the sequential paradigm. The paradigm of parallel processes was early introduced [Dijkstra71] to cope with such complex processing and it should be a must to apply it. However, many today's operating systems only provide it for the application level, where it was necessary for users to virtually achieve sequential behavior.

Structuring different unrelated streams of instructions into multiple activities enclosing independent processing, provides a higher abstraction of processing, decouples the processing of some tasks from the real multi-switched processing at the CPU level. A single process gives the illusion of a single unaffected processing or running on an idealized virtual processor. This is widely applied on top of most operating systems today. It is also beneficial for the structure of the operating system itself and must be applied there.

Machines with multiple CPU's fit very well to this approach, there is no essential difference in control between unpredictably interleaved processing on one CPU or processing on multiple CPU's. There is of course a difference in time.
The term "activity" is preferred here in respect to the ambiguous use of the term "process" in many systems. An activity just means the bare processing and its control. There are no other subjects related, as contexts and address spaces.

Are virtual processors real carriers of an activity? Viewing a system at this level, yes they are. A virtual processor is an example of a new quality thing that really exists at a certain level of a system and which is constituted by infrastructural objects, which are representations in the infrastructure that keep states, especially for (1)-control, and which also give an identity to them. They have behavior which is the activity for the instance area.
An infrastructure that provides multiple activities must also provide services for their control for instances to enable them to communicate or to coordinate. For these services activities must also be identifiable for instances.

2.4.2 Objects and Activities

After the nature of activities was detailed, their relations to objects should be discussed next. Objects are primarily things to which some "action is directed". They are primarily passive within the quality they are viewed. Objects provide services for other objects, and services have to be performed some way. That is the point where activities come in. Not an object does anything, but an activity. Putting the two things objects and activities of one level into relation, four major combinations are reasonable:

1. **n objects, 1 global activity:**
   sequential, (like C++),

2. **n objects, m global (unrelated) activities:** (the most general form)
   parallel, (multithreaded C++, as applied in PANDA),

3. **n objects, 1 activity each:**
   inter-object parallelism, such objects have an own activity and are often called *active objects*, *actors* or *concurrent objects* (ABCL/1),

4. **n objects, m activities each:**
   inter- and intra-object parallelism or active objects with multiple activities, to possibly perform multiple services simultaneously or doing some internal processing (BirliX-instances), the number of activities may be fixed or may dynamically change on demand, such as a new activity might be created when a service is to be performed.

The first is unappropriately for operating systems due to their multiple activities. The second is unsuitable because of complexity, although this is the most general form. This form closely corresponds to the classical separation of processes and resources in operating systems [Kalfa88]. The other imply restrictions to achieve a better structure. The advantage of the third and fourth form only arises in combination with object-orientation. Objects do not only encapsulate their private state and the descriptions of behavior (methods), they also encapsulate activities. This means, the activity(-ies) of an object does only affect the local state of the object, which is in contrast to the second form, where any state of any object may be affected. This reduces the occurrence of critical sections and incidentally encourages monitor-like encapsulation in objects. It is obviously that a good structure forces to have only one of the above forms within a certain quality of objects. The flexibility of having more such forms could be integrated by creating different qualities, which naturally applies to the multi-level architecture.

Objects encapsulate activities performing services. (2)-control is given for activities by the methods of objects. Performing services also requires interaction among objects [Wettstein93] that primarily concerns two basic mechanisms:

control, for the coordination of two or more activities, that may be centralized in the infrastructure ((1)-control) or may be decentralized among the involved activities in their (2)-control, and

data shipping, either via commonly accessible memory (cooperation) or via a channel with no common memory (communication), both of them need control.

A reduced form of interaction could only affect control (signaling). Due to centralizing common properties in object-oriented systems, the infrastructure should provide means of (1)-control for object interaction. As well as objects encapsulate internal properties, they may also encapsulate internal activities. From an outside view there is only object interaction visible, which could be seen as "meta-activities". This is only possible for the second and third form mentioned above which is in connection with object-orientation. Regardless of object internal activities, there are two primary forms of meta-activities:

- *procedural* – or transferring a meta-activity to another instance,
  the requestor stops while performing the service, but the service may be served concurrently to other services at the server, and of course concurrently to other objects,
- *fork* – fork a new meta-activity,
  the requestor continues to work after its request, and may later be notified that the service has done,
  join – synchronize previously forked meta-activities and reunite them.

2.4.3 Qualities making Levels

Qualities distinguish levels in the architecture which is proposed here. Qualities generally define the kinds of objects, their constitutes and their global behavior. The *quality* is mainly determined by *general and common properties for all objects*, which are mostly laid down in the infrastructure. This is encouraged by object-orientation to centralize such common properties in the infrastructure. The infrastructure determines:

- what is identifiable, and therefore considered as objects in the instance area,
- the representation of objects, thus their existence, states, and behavior,
- how activities apply to objects, and
- what kind of interaction is possible.

These are also the main points making qualities. Object systems cover such properties by their individual "object model". Two examples should be briefly discussed for that, the object models of *C++* and *BirliX*.

The run time model of *C++* defines objects as portions of memory representing their states. Identification is thus based on memory addresses. The property of memory (storage class) determines the lifetime of objects, whether they are static or automatic objects, or dynamically allocated (`new`-operator) and later explicitly released. No object lives longer than the application runs, because there are just representations in main memory, which is cleared after the exit. All objects share the same address space. There is only one activity within a system of C++-objects, that reduces interaction to procedure calls. The infrastructure for this is quite simple. There is only a little run time system, that makes sure that constructors and destructors of static or dynamically allocated objects are initially and finally called. There is also some dynamic memory allocation to be maintained, but that's all. This model is tailored to the current hardware infrastructure, especially to the addressing scheme and the call-facility of the CPU. This makes it very efficient and suitable for fine granularities of objects.

The object model of the *BirliX* operating system suites to coarse granularities of objects, such as a directory service or a file service. BirliX supports objects interacting across different machines. Identification of objects can therefore not be bound to memory addresses. BirliX uses unique identifiers (uid's), which are 64-bit strings and are generated by the system at the creation of an object. BirliX objects primarily exist on stable storage (persistent representation which is called a "team"). Only for the purpose of processing to perform some service the object comes into main memory with a private address space and multiple activities. Each service is performed by a newly created activity (thread). A team becomes "active" for this purpose. Checkpoints may be made, initiated by the object, which are representations of the object state, including its processing state. These checkpoints may later be recovered to reset a failed transaction or may be transferred to another machine of equal type and might be recovered there (object migration).

The two examples show, what is meant with qualities here. Qualities are made by such fundamental properties of the infrastructure. At concrete systems there is mostly just one model supported, which of course cannot cover all the needs of different kinds of applications. That's why multiple *object model's* or *qualities* should be applied to complex systems, such as operating systems. Qualities are investigated in more detail in the next two sections.

2.4.4 Qualities of Representations

The kind of representations of objects in the infrastructure mostly determines their properties of creation, existence and destruction. The representations themselves are objects within the infrastructure. The existence of the representations constitutes the existence of objects in the instance area. Objects are considered as software here, that only can exist in any kind of memory to keep the information of the state and (1)-control of the object. There are different kinds of memory implying different properties for objects.
An automatic C++-object does only exist if the activity runs the code within the block the object was defined in. It is dynamically created at the entrance of the block and destroyed after its leave. The reason for this property lies in the infrastructure, where representations of such kinds of objects are stored on the run time stack, which quickly allows the allocation and deallocation of memory for representations by simply in- or decrementing the stack pointer. Static objects exist from the start of the application until its end, because the memory the representations are stored in has this property. If representations are written to disk, they survive the end of a application process and thus the object survives as well (property of persistence).

2.4.5 Qualities of Activities

Up to now, activities were considered in a real pure sense, just as streams or flows of processing control instructions. There was no context involved, that however distinguishes for example interrupt activities from UNIX-process-like activities.

Three main points may be found to distinguish qualities of activities:

granulatity: What is the state set the activity affects in the instance area?
Encapsulating an activity means to reduce this set to only the state set of the associated object. If there is no such encapsulation, the set potentially includes all state sets of all objects in the instance area.

context: What state set is necessary to represent the activity itself in the infrastructure?
This set is needed to resume an activity if it was previously switched off. It is of special interest for activities that virtually share the same resources but need them exclusively, thus the infrastructure needs to multiplex them among the activities. The state set contains the values of all such shared resources. Another point of interest here is to possibly ship the set to another location and recover it there.
If there are multiple activities virtually running on one CPU, the state set includes at least the values stored in the CPU registers. If there are other exclusive resources shared, their states add to the set, such as the address translations of the shared MMU in the case of UNIX-like processes with separate address spaces.
The extend of this set is commonly termed "weight", there are light weight activities that have no exclusive resources, except the CPU, in contrast to heavy weight activities, that share much more exclusive resources, such as MMU translations.

global states: Global states concern (1)-control in the infrastructure. There are at least two such states: `active` and `stopped`. The last state may furtherly be differentiated in any detail, such as ready, or waiting for something and so far.

There are a variety of possible combinations. In section 4 an example for such a special kind of activity is investigated, so called interrupt processes or *iproc's*.

Representations and activities are the two main things that are generally important for making qualities of objects. Other properties, that were not just considered here, also set up qualities, such as protection at different levels, guaranteed qualities of some services, failure tolerance, distribution or something like that, but they are not generally necessary.

3 The Architecture of a Multi-Level object-oriented Operating System

An operating systems provides an abstract machine for instances running on it. This separation of infrastructure and instance area should be applied at different levels to structure an operating system up from the lowest level, which forms the interrupt system at current machines. The goal of this work is to isolate such levels with a clear internal structure and compose these levels to provide the functionality of an operating system. This structuring is more flexible than

the traditional multi-layered structuring, because multiple (optimal) run time environments or infrastructures could be provided for disjunct sets of instances at the same level.

It is an architectural decision whether to place functionality inside the infrastructure or outside in the instance area. Universal operating systems tend to place many things inside their kernels (infrastructure) to cope with a wide range of application requirements. μ-kernels follow the opposite direction by placing as much as possible in the instance area. Here, multiple levels are promoted, as already found in some existing systems. For instance, there are run time environments that adapt general operating systems environments to the needs of certain languages, such as extensions for distribution or concurrency, as shown in section 2.1 above. These are examples for partially multi-level approaches. Here, multi-level structuring is introduced as a general principle, not just a vehicle for insufficiencies of existing run time environments.

It is not aimed to provide a mass of universal infrastructures that cover any possible requirements, but to provide a structure, where it is *possible to easily integrate new infrastructures* on demand. This is an extended understanding of *reconfiguration*. Reconfiguration is quite complicated in current systems, because of their monolithic structure. It is mostly only possible at the source code- or link-level, it is hardly supported at boot time, or even at run time. Building new infrastructures for certain sets of instances should be natural however and therefore be very supported! This primarily means the facility of *putting new descriptions of new objects into infrastructures*, that later instances could be made of. This is not possible at most systems today beyond the source code- or link-level, and even there it is difficult. However, dynamic reconfiguration is already possible for some UNIX-systems concerning device drivers. AIX for instance, allows to load the object file of a new driver into the kernel and to make it available at run time.

If object-orientation is not reduced to the programming language level, where the object structure is mostly lost after compilation, this is much easier. The μ-kernel approach gives an example. Objects are not instances of C++-classes, but dynamically creatable processes at run time that are performing services. However, C++ may be used to produce the descriptions (.o-files). But this is another quality of objects. In *MACH*, the descriptions for representations of such objects (code, initialized data, and sizes of segments) are assembled at boot time (system configuration). The descriptions of any later created object must be loaded into the infrastructure at this time. It must be fully known at this time. For example, *MACH's* external pager contains the policy for paging, which possibly could be replaced by simply starting another server with a different strategy at boot time. Other systems require relinking or even recompiling for that purpose.
Following this direction consequently, it should be possible to load new descriptions down to the infrastructure even at run time. Making this possible across all levels it allows to set up new kinds of infrastructures or to adapt present infrastructures if it is necessary.

Current hardware mostly supports monolithic systems (common address space, procedure calls). This makes reconfiguration difficult. One way that is suitable for objects of larger granularity is to provide them with separate address spaces by their infrastructure and to base interaction on exchanging messages. This is applied to current μ-kernels. If a message is sent to one object or if it is redirected to another one after reconfiguration does not affect the requestor in any way. The infrastructure fully provides it. This feature was exploited for introducing dynamic adaptability in the BirliX operating system [Sonntag94].
But if there is procedural interaction in a monolithic kernel, the infrastructure relates to the fixed CPU-"hardware". The identification and addressing of objects is based on the addresses

in the call instructions at the side of the requestor. Changing interaction by replacing objects directly affects the representations of all involved objects, because the relocation of all such addresses is necessary.
Another way to obtain reconfigurability is to make use of indirect procedure calls and just manipulating the pointers, as C++ allows it for calls to virtual functions.

There are also virtual machines immediately interpreting the source code (e.g. Smalltalk) where it is easily possible to exchange or add new types of objects, but these systems are not suited to operating systems, because they rely on virtual machines that themselves are software that must run somewhere.

The discussion shows, that reconfigurability is also possible in a monolithic object system, if the infrastructure provides a relocation mechanism for the code. Monolithic object systems suite better to the current hardware infrastructure, mainly the CPU and are therefore very efficient. Object systems of instances with separate address spaces are much easier to handle for reconfiguration, but are less efficient due to their "unnatural" interaction based on messages that must be provided by the infrastructure and is not supported by current CPU's. The sending of messages does not only mean the shipping of some data, it also includes process control. The receiving process must be awakened, scheduled, and context-switched (including the address space) before getting the message. The same rolls back for delivering the answer to the requestor. Hiding this by some RPC-cosmetics makes this only invisible at the program level, but doesn't effect what real happens on the machine. It could be optimized to some degree but there will always remain much more effort compared to simple procedure calls in a monolithic kernel.

4 Structuring the Interrupt System by introducing *iproc's* – the level 0 of the recursive approach –

The *interrupt system* forms the bottom layer of any operating system. Interrupts are caused by signals among activities running on the various active hardware units. To avoid polling such signals the interrupt system provides interrupting the programmed flow of control to notify the CPU-activity on the receipt of a signal. The interrupt mechanism is a means of (1)-control of the CPU. The traditional procedure-oriented view of handling interrupts has some peculiarities:

- interrupts are not predictable which leads to nondeterministic switches in the flow of (2)-control of the CPU, such switches are not pictured in the program,
- critical sections can occur with the need to identify and to coordinate them,
- classes of interrupts and their different properties are not distinguished leading to the misunderstanding that interrupt service routines are just special kinds of procedures,
- the interrupt system of current CPU-architectures is processor-oriented, but coordinating activities are abstractions of processes in operating systems today.

The interrupt system forms the infrastructure for the software controlled activities on a CPU. Its ability to preempt control makes multiprocessing possible. The coordination of critical sections is achieved by masking or disabling interrupts to prevent switching, by the arbitration of interrupts

and by the setting of interrupt levels. The interrupt system was introduced at the very beginning of computers and has not been changed in principle over the years. In the early years of computers programmers thought in terms of processors, not processes. This is reflected in the interrupt system of today's computers. The interrupt system is processor-oriented, not process-oriented. In [Gittinger92] a proposal is made to support process-orientation at the CPU-level and to apply signaling of processes replacing interrupts, but it relies on a new type of CPU.
The interrupt system of today's CPU-architectures is not only used for coordination with other active units in the system. One can identify two basic categories of interrupts:

class A: *asynchronous*, issued by other processors to notify the CPU-activity (I/O-controllers, clock unit, DMA, etc.),

class S: *synchronous*, issued by the (2)-control of the CPU itself,

- **class SE:** *exception*, unprogrammed switch in the flow of instructions according to an exceptional condition on one processor (division by zero, page fault, etc.);
- **class ST:** *trap*, programmed switch on one processor or indirect procedure call to an interrupt service routine, often with the side effect of raising the processor level as the only protected way to enter the kernel.

It is always an interesting question whether handling interrupts is procedural or process-oriented. Running an interrupt service routine starts always at the same entry point and is processed until the end. An interrupt service routine does not keep processing state while not being active. That's all procedural behavior. But, only *class ST* interrupts can be viewed fully procedural. As soon as calls to service routines are no longer part of the program, nondeterminism arises. There is no longer procedural semantics. Critical sections may occur that must be coordinated. This concerns *class A* and partially *class SE* interrupts. However, unprogrammed nondeterministic switches and the occurrence of critical sections fit well to the view of multiple activities of a special quality, which are called interrupt processes or *iproc's* here.

4.1 The Design of an Interrupt System based on *iproc's*

In accordance with the above discussion on structuring the interrupt system by *iproc's*, the following design goals could be set up:

- the handling of an interrupt is done by an *iproc*-activity that performs the code of the interrupt service routine, each interrupt causes the creation of a new *iproc* with the appropriate service routine assigned, thus, there may be multiple *iproc's* performing the same code in the case of interleaving interrupts,
- *iproc's* are instances, their services are the handling of one interrupt each, the infrastructure for these instances is given by the interrupt handling facility of the CPU and few representation data,
- *iproc's* must be of an "ultra light" weight with a minimum of private state to make them as efficient as conventional handling interrupts, this is primarily achieved by the preallocation of most resources and easily assigning them when an *iproc* is created,

- the mechanism of controlling *iproc's* and the control policy should be separated (scheduling), the processing of interleaved interrupts should only depend on the policy,
- the instance area of *iproc's* must provide the infrastructure for higher level instance areas, and must consequently contain representations for their instances as well,
- critical sections are encapsulated within *iproc's*, as discussed for activities in objects in section 2.4.2.

Running interrupt service routines by *iproc's* requires dedicated properties to make their handling as efficient as the traditional invocation of service routines. For this purpose most resources are statically preallocated. Such resources are the memory for the private stack and representation data. There is an array of entries, one entry for each *iproc*. When an *iproc* is created at the occurrence of an interrupt, an entry must be allocated, which is easily done by keeping pointers to free entries in a separate stack-organized data structure. A new entry is thus allocated by performing a pop on that structure. The number of entries is fixed and there is no dynamic data structure to achieve a deterministic allocation time. The identification of the interrupt (interrupt number) selects the interrupt service routine (code) to be run by the *iproc*. These are really cheap operations necessary to create an *iproc*.
Next, the policy instance selects the next ready *iproc* and this is activated by setting simply the SP to the private stack of the *iproc*, which is associated with the representation entry, and the PC to the service routine.
After the *iproc* has finished handling the interrupt it is released by pushing the pointer of its representation entry back to the free stack and marking this entry as invalid for the policy instance.
Only in the case of interleaving interrupts more than one *iproc* exist at the same time, actually requiring the selection by a policy. In most cases there is only one *iproc* (immediately after an interrupt), which also makes the policy very efficient.

This way, creating and switching *iproc's* is really cheap by simply manipulating CPU-registers (SP,PC and some other). Nothing else actually happens at conventional interrupt handling. The additional overhead for maintaining representations of *iproc's* and for the policy could be minimized to a negligible quantity.

To summarize the idea, three steps are outlined that happen at the occurrence of an interrupt:

1. primarily CPU-oriented handling of an interrupt (infrastructural level):
   - preempt the currently running activity by saving its preemption state, but not to a common interrupt stack, instead to a private location associated with the representation of the preempted activity,
   - identify the interrupt (interrupt number, a generated sequence number for uniqueness (counter), and optional arguments such as the faulting address of a page fault interrupt),
   - create an *iproc* as described above (allocate an entry and plug in the appropriate SP and PC values),
   - return from the basic CPU-interrupt cycle, but don't resume the previously preempted activity, instead transfer to the control instance for *iproc's* (see figure 2), this is similar to the mechanism of *continuations* discussed in [Draves91].

2. the control instance calls the policy which decides on the further transfer of control,

3. the control instance transfers the control to the selected *iproc* that runs the code of the service routine and returns to the control instance after it has done this.

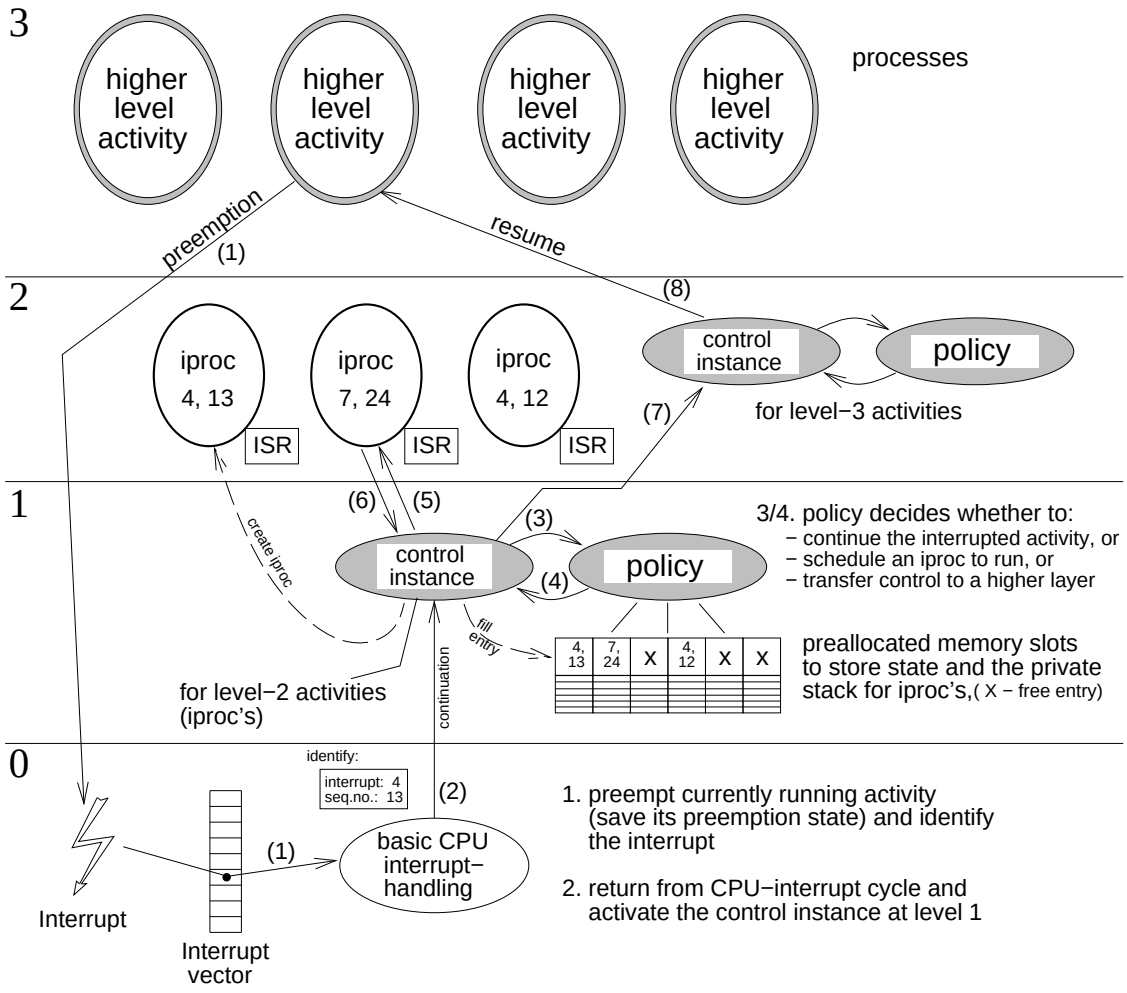

Figure 2: *iproc's* as Instances handling Interrupts

It should be stressed, that the control instance runs completely outside the interrupt cycle of the CPU keeping further interrupts enabled. Interrupts are disabled only to avoid critical sections within the control instance itself. But this concerns only a few instructions. All other coordination of critical sections of *iproc's* is done by notifying the control instance not to switch in the meanwhile, but keeping the acceptance of further interrupts by creating new *iproc's* and delaying their processing. Dissabling interrupts of the CPU has the same effect, but multiple interrupts of the same type may be lost, because they are not stored in current CPU's.
Multiple instances of *iproc's* can exist when handling of an *iproc* is interrupted itself in the case of interleaving interrupts. Similarly to preempting other kinds of activities, the preemption state of a running *iproc* must be saved in a location that is statically reserved in the representation of the *iproc*. The preemption state consists of the register set of the CPU.

That's a different approach compared to [Hills93] using cyclic processes, each for one interrupt type which block in a `wait_on_signal` condition and wake up if an interrupt has occurred. Interleaved handling of interrupts of the same type is not possible with this scheme. Concerning the costs there in no substantial difference between creating a new or awaking an already existing *iproc* at the occurrence of an interrupt. However, the solution of *iproc's* is more general. Another discussion for structuring interrupts is in [Chrobot94].

4.2 Advantages

Abstraction always causes costs. But limiting costs to a minimum can prove abstractions to be applicable. The presented scheme of a process-oriented interrupt handling is based on dedicated interrupt processes called *iproc's* and is more general and equally efficient compared to conventional solutions of handling interrupts.
There are several advantages which are cheaply to achieve by *iproc's*. Interleaving of interrupts is well suited to *iproc's*. There is just more than one *iproc* at a time, each representing an interrupt. No interrupt may potentially be lost. A policy decides which to run. The policy is free in principle and no longer fixed by ordered interrupt frames on a shared interrupt stack.
Real-time constraints could be integrated into the policy as well. The implementation of our basic interrupt handling and its control instance avoids dynamic data or algorithms, such as loops depending on dynamic values of variables. By this way the duration of elementary operations is always predictable which is essential for real time policies. Another feature enhancing real time constraints is that phases with disabled interrupts only concern few instructions within the control instance. All coordination of critical sections is otherwise fully under the software-control of the control instance.

5 Status and Future Work

The proposed approach of multiple levels of infrastructures and instance areas is applied to an experimental operating system called *OMEGA*. It is set upon bare hardware, not in a simulation environment, because many effects which are under (future) investigation, such as distributed shared memory (DSM) on losely coupled workstations at the level of pages, are only possible to realize in that "hard"-environment.

There are currently two hardware platforms supported, a CADMUS workstation based on 2 m68020 CPU's and 486-PC's running in protected mode. The system development is done under UNIX using the GNU C++-compiler. The level 0 infrastructure for the interrupt system has been successfully completed for both platforms. There are currently device drivers for the terminal, the clock and the disk successfully been integrated into *iproc's*. It turned out that there is just a negligible run time overhead for *iproc's*, when a simple policy is applied, such as static priorities for *iproc's*. This statement is based on comparing the number of machine instructions to run, which are a measure for efficiency, because there are no repeated sequences in this code.

The next step will be to set up a well structured level 1 infrastructure for the basic device drivers for the terminal, the disk and the network. The level 1 infrastructure is itself realized within the instance area of *iproc's*. After the device drivers will have successfully been integrated, the level 2

infrastructures are to be investigated for file-system-like services and run time environments for applications.

The DSM direction mentioned above aims to provide virtually access to memory pages at losely coupled remote machines at the level of pages, which makes such accesses fully transparently at the level of machine instructions. The CPU just carries out the programmed accesses to virtual memory. What the contents of such memory is, depends on the mapping which is done by the infrastructure. The infrastructure could contain instances handling a protocol to keep the contents of shared pages at different machines consistently. Such a facility offers a large variety of properties, distributed applications could use and avoids the need for the conventional proxies. The assumption is, that accesses to remote objects are rarely compared to accesses to local objects, and thus accesses via immediate procedure calls to local objects should be preferred. There is no additional overhead for the proxy mechanism. Accesses to remote objects are triggered via page faults. They are rarely and thus the overhead of a page fault seems acceptable.

Although the experimental operating system is still under investigation and development, multi-level structuring in connection with object-orientation has proved to be suitable to homogeneously structure running systems across different levels, from the hardware-level up to the level of applications.

References

[**Accetta86**] Accetta,M., Baron,R., Golub,D., Rashid,R., Tevanian,A., Young,M.: *'Mach: A New Kernel Foundation for UNIX Development'*, Proc. Summer 1986 USENIX Conference, pp.93-112, 1986.

[**Assenmacher93**] Assenmacher,H., Breitbach,T., Buhler,P., Hübsch,V., Schwarz,R.: *'PANDA - Supporting Distributed Programming in C++'*, Technical Report, University of Kaiserslautern, Dep.of CS, 23 p., 1993.

[**Bershad88**] Bershad,B.N., Lazowska,E.D., Levy,H.M.: *'PRESTO: A System for Object-oriented Parallel Programming'*, Software-Practice & Experience., Vol.18, No.8, pp.713-732, August 1988.

[**Buhr92**] Buhr,P.A., Ditchfield,G., Stroobosscher,R.A., Younger,B.M.: *'μC++: Concurrency in the Object-oriented Language C++'*, Software-Practice & Experience, Vol.22, No.2, pp.137-172, February 1992.

[**Chrobot94**] Chrobot,S., Stras,A., Stras,R.: *'Needles and Links or Dealing with Interrupts'*, Technical Report, Kuwait University, Department of Mathematics, Safat, 11p., 1994.

[**Cooper87**] Cooper,E.C., Draves,R.P.: *'C–Threads'*, Carnegie Mellon University, Technical Report, CS-88-154, 16p., July 1987.

[**Creasy81**] Creasy, R.J.: *'The Origin of the VM/370 Time-Sharing Systems'*, IBM Journal of Research and Development, Vol.25, No.5, pp.483-490, 1981.

[**CSA90**] : *'CSA: Communications Systems Architecture – Architectural Description'*, MARI Applied Technologies Ltd, 236p., 1990.

[**Dijkstra68**] Dijkstra,E.W.: *'The Structure of the THE - Multiprogramming System'*, CACM, Vol.11, No.5, May 1968.

[**Dijkstra71**] Dijkstra,E.W.: *'Hierarchical Ordering of Sequential Processes'*, Acta Informatica, Vol.1, pp.115-138, 1971.

[**Draves91**] Draves,R.P., et al.: *'Using Continuations to Implement thread Management and Communication in Operating Systems'*, Proc. 13th ACM Symposium on Operating System Principles, October 1991.

[**Gittinger92**] Gittinger,J., Krüger,U.: *'Prozeßsignalisierung statt Unterbrechungen'*, (in German), Interner Bericht 11/92, Universität Karlsruhe, Fakultät für Informatik, 37 S., Mai 1992.

[**Golub90**] Golub,D., Dean,R., Forin,A., Rashid,R.: *'Unix as an Application Program'*, Proceedings of the USENIX Summer Conference, June 1990.

[**Graupner94**] Graupner,S.: *'Coroutinen und preemptive Threads in C – Mechanismen und Realisierung'*, (in German), Bericht, TU Chemnitz, Fakultät für Informatik, 50S., Februar 1994,

[**Härtig90**] Härtig,H., Kühnhauser,W.E., Lux,W., Reck,W.: *'Architecture of the BirliX Operating System'*, GMD, St.Augustin, 6 p., März 1990.

[**Hills93**] Hills,T.: *'Structured Interrupts'*, Operating Systems Review, Vol.27, No.1, pp.51-68, January 1993.

[**Kalfa88**] Kalfa,W.: *'Betriebssysteme'*, (in German), Akademie-Verlag, Berlin, 400 S., 1988.

[**Kalfa92**] Kalfa,W.: *'Proposal on an External Processor Scheduling in Micro-Kernel based Operating Systems'*, Technical Report, TR-92-028, ICSI, Berkeley, 14p., May 1992.

[**Kiczales91**] Kiczales,G., Rivieres,J., Bobrow,D.G.: *'The Art of the Metaobject Protocol'*, The MIT Press, 335p., 1991.

[**Kleinöder92**] Kleinöder,J.: *'PM Systemarchitektur'*, (in German), Technical Report, TR-14-14-92, FAU Erlangen-Nürnberg, Oktober 1992.

[**Maes88**] Maes,P., Nardi,D.: *'Meta-Level Architecture and Reflection'*, North-Holland, 355p., 1988.

[**Marty89**] Marty,R.: *'Von der Subroutinentechnik zu Klassenhierarchien'*, (in German), Berichte des Instituts für Informatik, Nr.88.04, Universität Zürich, 41 S., 1989.

[**Müller91**] Müller,K.: *'Techniken der objektorientierten Programmierung'*, (in German), GUUG-Nachrichten, Springer Verlag, Heft 25, S.19-33, April 1991.

[**Müller92**] Müller,K.: *'Realisierung verteilter Objektstrukturen mit C++'*, (in German), Bericht, TU Chemnitz, Fakultät für Informatik, 83 S., 1992.

[**OMA92**] : *'Object Management Architecture Guide'*, OMG Document No. 92.11.1, Revision2.0, OMG, 98p., 1992.

[**Parnas72**] Parnas,D.L.: *'On the Criteria To Be Used in Decomposing Systems into Modules'*, CACM, Vol.15, No.12, pp.1053-1058, December 1972.

[**Ritchie74**] Ritchie,D., Thompson,K.: *'The UNIX Time-Sharing System'*, CACM, Vol.17, No.7, pp.365-375, July 1974.

[**Rozier88**] Rozier,M., Abrossimov,V., Boule,I., Gien,M., Guillemont,M., Herrmann,F., Kaiser,C., Langlois,S., Leonard,P., Neuhauser,W.: *'Chorus Distributed Operating Systems'*, Computing Systems, Vol.1, No.4, pp.305-370, Fall 1988.

[**Russo91**] Russo,V.F.: *'An Object-Oriented Operating System'*, Ph.D. Thesis, University of Illinois, 154p., 1991.

[**Schill93**] Schill,A.: *'DCE: The OSF distributed computing environment'*, Proc., LNCS No.731, 283 p., Springer, 1993.

[**Schröder93**] Schröder-Preikschat,W.: *'A PEACE Case Study'*, Technical Report, TR-93-020, ICSI, Berkeley, April 1993.

[**Shapiro89**] Shapiro,M., Gourhant,Y., Habert,S., Mosseri,L., Ruffin,M., Valot,C.,: *'SOS: An Object-Oriented Operating System - Assessment and Perspectives'*, Computing Systems, Vol.2, No.4, pp.287-337, Fall 1989.

[**Snyder93**] Snyder,A.: *'The Essence of Objects: Concepts and Terms'*, IEEE Software, Vol.10, No.1, pp.31-42, January 1993.

[**Sonntag94**] Sonntag,S.: *'Adaptierbarkeit durch Reflexion'*, (in German), Dissertation, TU Chemnitz, Fakultät für Informatik, 103S., Februar 1994.

[**Stoustrup90**] Stoustrup,B.: *'The Annotated C++ Reference Manual'*, Addison-Wesley, 447p., 1991.

[**Szyperski92**] Szyperski,C.A.: *'Insight ETHOS: On object-orientation in operating systems'*, Ph.D. Thesis, ETH Zürich, 232p., 1992.

[**Tanenbaum90**] Tanenbaum,A.: *'Betriebssysteme: Entwurf und Realisierung (Teile 1/2)'*, Hanser-Verlag, München, 322 S., 1990.

[**Web88**] : *'Webb Webster's New World Dictionary'*, Third College Edition, Simon & Schusters, Inc., 1988.

[**Wegner90**] Wegner,P.: *'Concepts and Paradigms of Object-Oriented Programming'*, OOP Messenger, 1(1), pp.8-85, 1990.

[**Wettstein93**] Wettstein,H.: *'Systemarchitektur'*, (in German), Carl Hanser Verlag München Wien, 514 S., 1993.

[**Wiatrowski92**] Wiatrowski,J.: *'Entwurf und Implementierung eines objektorientierten Betriebssystems'*, (in German), Diplomarbeit, TU Chemnitz, Fakultät für Informatik, 1992.

[**Wirth88**] Wirth,N.: *'Oberon'*, Software-Practice & Experience., Vol.18, No.7, pp.671-690, July 1988.

[**Yokote92**] Yokote,Y.: *'The Apertos Reflective Operating System: The Concept and its Impementation'*, Proc.of OOPSLA'92, October 1992.

[**Yonezawa86**] Yonezawa,A., Briot,J.P., Shibayama,E.: *'Object-Oriented Concurrent Programming in ABCL/1'*, Proc.of OOPSLA'86, ACM, September-October 1986.