Paramecium: An extensible object-based kernel

Leendert van Doorn
Philip Homburg
Andrew S. Tanenbaum

Vrije Universiteit
Amsterdam, The Netherlands

ABSTRACT

In this paper we describe the design of an extensible kernel, called *Paramecium.* This kernel uses an object-based software architecture which together with instance naming, late binding and explicit overrides enables easy reconfiguration. Determining which components reside in the kernel protection domain is up to the user. An certification authority or one of its delegates certifies which components are trustworthy and therefore permitted to run in the kernel protection domain. These delegates may include validation programs, correctness provers, and system administrators. The main advantage of certifications is that it can handle trust and sharing in a non-cooperative environment.

1. Introduction

Traditional kernel design tends to get in the way of contemporary application demands. For example, diverse application areas like multi-media, wide area communication, and parallel computations have very different operating system requirements. Providing a single virtual machine to support all these demands, either results in very large and complex systems that provide the necessary support but at a high cost in added complexity and loss of efficiency, or in application specific and usually very rigid systems.

In this position paper we propose a design for a highly dynamic kernel, which enables us to build application specific operating systems without the loss of generality. Central in this design is a common software architecture for operating system and application components. We use this architecture to construct a toolbox of components. These components can, with some minimal kernel support, be configured dynamically to reside either in the kernel or in the application's address space. The system uses late binding and dynamic loading to instantiate components at run time.

Determining which components reside in user and kernel space is up to the user. An authority certifies which components are trustworthy and are therefore permitted to run in the kernel address space. Each component contains a certificate that is validated by the kernel by means of a simple security architecture.

The certification authority will usually delegate its authority to subordinates. These include system administrators, experimenters who need complete control over a particular machine, and programs. Multiple delegates can also be used to form an escape hatch: if one subordinate fails to certify a component another can be tried. This is especially useful when programs are used for certification, as these may have a limited application domain.

This approach generalizes the direction taken by two other projects, the Exo-kernel [2] and SPIN [1]. These projects allow any application to insert user code into the kernel's protection domain and provide elaborate software technology, such as restricted, type safe languages† and sandboxing‡ [11], to prevent it from causing harm.

Although this software approach is certainly useful and might be sufficient for many application, it does not suffice for components that manipulate *trust* or *sharing* in non-cooperative environments. For example, inserting application components for fast protocol processing into a shared network device driver is close to impossible. Software verification of the component cannot easily reveal packet snooping, hence it is hard to detect a breach in kernel security. Certifying the component by a trusted authority and verifying this at load time solves the trust problem and is more efficient because of the absence of run time checks.

Because we use the same software architecture for building application programs, programmers can use the same mechanisms to control which

version of a component is used. Constructing interposing agents [3] is trivial, enabling the construction of powerful monitoring tools.

To validate this software architecture and certification technique we are building a prototype kernel, called *Paramecium*, which is intended to provide support for parallel programming. We have chosen this area because an associated group, involved in parallel programming research, needs better and finer grained control over the machine's hardware. They are also able to provide us with many applications, which we can use to test our design.

The remainder of this paper is divided into five sections. Section 2 discusses the necessary software architecture for building components, which is followed by Section 3 describing a set of standard abstractions provided by the Paramecium kernel. Section 4 describes the certification concepts, followed by section 5 describing related work.

2. Software Architecture

In order to support the toolbox approach we have designed a simple, programming language independent architecture that provides object instances and interfaces as its main abstractions. Both operating system and application components are written according to this architecture. This allows their components to be interchanged.

In our architecture an *object* is conceptually a collection of methods and instance data. Each object exports one or more named interfaces. This provides support for evolving and generic interfaces. For example, adding a measurement interface to an RPC object does not require recompilation of its users, since the RPC interface itself does not change.

An interface is a set of methods, state pointers and type information. Objects can be operated on only through the methods in the interfaces they export. Objects are relatively coarse grained. They can contain operating system components such as a scheduler, IP layer, or a device driver, or application components such as memory allocators or matrices. To support code sharing the architecture supports method delegation.

Besides objects and interfaces the architecture also supports a third concept: composition. A composition is an ordinary object composed of other object instances. Composition is to objects what objects are to data: an encapsulation technique. For example, the Paramecium kernel is a composition, composed of objects that manage interrupts, user contexts, etc. Note that composition can be applied recursively.

Objects are usually loaded dynamically on demand. Objects in a composition, however, can be loaded statically or dynamically. That is, the composition is either created at link time or run time. The latter is the most common form of object composition since it allows for the composing objects to be replaced by new instances. Static composition is currently only used for building the resident part of the kernel.

Each object has its own instance name and is registered in a hierarchical name space together with its object handle. This name is used by other objects to bind to it. Standard operations exist to bind to an existing object, load one from a repository, and to obtain an interface from a given object handle.

The main advantage of using a name space for object instances is its ability to be reconfigured. For example, building an interposing agent [3] for a network device, `/shared/network`, consists of building an interposing object (i.e., one that exports a superset of the original object's interfaces, reimplements those methods it sees fit and forwards the others to the original object) and replace the object handle in the name space. All further lookups for `/shared/network` will result in a reference to the interposing agent.

The name space is usually inherited from a parent, i.e., the object that created it. Each object, however, can provide a set of overrides which allows it to locally reconfigure its name space: that is, control the child objects it will import. This mechanism allows the programmer to control and specify the components its application will use. Apart from being useful for debugging, together with interposing objects it enables the construction of powerful monitoring tools.

A potential drawback of an object based software architecture, like ours, is performance. This is mainly caused by the method invocation overhead costs and the introduction of additional software layers. Even though a method invocation is usually just a procedure call, these tend to be expensive on our target hardware (SPARC [5]). Still, we expect the overhead to be relatively low because our objects have a relatively large grain size. We are, however, contemplating run time inline techniques in case this might turn out to be a bottleneck.

3. System Architecture

The Paramecium system architecture consists of a nucleus and a repository of system components. The nucleus is a protected and trusted component which implements only those services that cannot be moved into the application without jeopardizing the system's integrity. All other system components, like thread packages, device drivers, and virtual

memory implementations reside outside this nucleus. Depending on the configuration they can be loaded in the kernel, or in the application's protection domain.

The nucleus provides four services, which all use a protection domain or context as their unit of granularity. The four services are:

- Processor event management

All processor events (traps and interrupts) are handled by this service. Components can register call-backs which are called every time a specified processor event occurs. A call-back consists of a context, and the address of a call-back function.

Processor events are usually redirected to the thread system to turn them into pop-up threads. Once interrupts are pop-up threads, they can block, and be scheduled just like any other ordinary thread. For efficiency reasons, we delay the actual creation of the pop-up thread by creating a proto-thread. Only when the proto-thread is about to block or be rescheduled do we turn it into a real thread. This allows us to provide fast interrupt processing of user code with proper thread semantics [10].

- Memory management

The management of virtual and physical pages, and MMU contexts, is done by the memory management service. Pages can be allocated exclusively or shared among different protection domains. Individual virtual pages can have fault call-backs associated with them. Cross-domain calls are implemented using per page fault-handlers and resembles the one described in [6].

Objects can be placed in separate MMU contexts. This is useful for isolating faults when debugging or when implementing active message like invocations.

The memory management service also provides I/O space allocation. Device drivers use this service to allocate I/O space and map in the device registers into their protection domain. I/O spaces can be allocated exclusively or shared, allowing device registers to be mapped privately and on-device buffers to be shared by other contexts.

- Directory service

The directory service implements the name space as described in the previous section. It provides functions for registering, unregistering, and binding of objects.

Cross-domain invocations are implemented using proxies. Importing an object from another protection domain, by means of the directory service, causes a proxy to appear. This proxy provides exactly the same set of interfaces as the original object, but each interface entry will cause a page fault when referenced. Control is then transferred to a per page fault handler which will map in arguments into the object's protection domain, switch context, and invoke the actual method. Return values are handled similarly.

- Certification service

Objects can be associated with a certificate that is validated by the certification service before mapping it into a protection domain. The certification service uses a message digest function, public key cryptography, and a trusted certification agent to validate credentials.

4. Certification

One of the most important functions of a kernel in a non-cooperating environment is to preserve the integrity of the system it runs on. Giving applications the ability to down-load arbitrary code into the kernel potentially violates this assumption. To overcome this dilemma we introduce a certification authority which determines whether or not a component is trustworthy enough to run in the kernel's protection domain.

The certification authority can choose to delegate its certification powers to subordinates. These may include programs, like type-safe language compilers or automated correctness provers, software test teams, system administrators, and even graduate students. These subordinates may be ordered in preference and provide an escape hatch if one of the subordinates fails to certify. For example, when the automatic program correctness prover decides that it cannot complete the proof, it might turn the problem over to the system administrator.

A certifier may take an arbitrary amount of time to validate a given component. It will usually be done off-line. This allows experimental object code provers like [13] that usually tend to take more time than, for example, sandboxing. This does not exclude on-line certification by the kernel.

The certification and delegation mechanisms are similar to those found in the Taos operating system where they are used for secure communication [4, 12]. In our system certificates include a message digest of the component so that it is impossible to modify the component after it has been certified.

The main advantages of certification are: It allows for a wide range of certification techniques and it is efficient. After a component's certificate is validated by the kernel it does not require any further software checks, unless these checks are required by the certification process. This is especially true when the certifier is a person who hand-checked the component, all run time checks can then be omitted.

Another advantage is that certification handles trust and sharing. Certified kernel components can include protocol stack implementations that are shared between multiple non-cooperating users, security modules, shared caches, etc. Trust and sharing are important notions in an operating system kernel that are hard to formalize and even harder to check automatically. The main disadvantage of certification is that it requires public key cryptography and key management.

5. Related work

Our work is similar to the Exo-Kernel [2] in that both systems allow applications to manage system resources and are intended to be flexible. The main difference is that Paramecium allows dynamic configuration of its operating system and application software components. The Exo-kernel provides this support in just one direction, from user to kernel protection domain. Down-loading application components is supported by the Exo-kernel by means of software protection techniques, like type safe languages and sandboxing, while Paramecium depends on certification. We believe that certification is more general. It encompasses the techniques used by the Exo-kernel, and, in addition, is able to deal with trust and sharing. Verifying a certificate at load-time obviates the need for run time fault checks thus allowing components to be more efficient.

The SPIN [1] project is aimed at providing extensible operating system support. Its main mechanism is the ability to down-load application code, written in a special type-safe language, into the kernel protection domain. It is straightforward to incorporate this technique in our certification system by delegating the certification authority to a trusted compiler for that language. Everything compiled by that compiler would then be automatically certified and safe to run in the kernel protection domain.

The name space concepts are somewhat similar to those found in the Spring operating system [7], although they are actually based on ideas from an early version of Amoeba [9]. Spring uses inheritance for interface evolution which allows extensions to an interface but not adaptations. Our method is more flexible.

Chorus [8] experimented with migrating user-level server code back into their microkernel to increase the efficiency. Our approach, however, is much more general. Many of the certification and delegation ideas are based on the authentication work by [4, 12]. that was used to secure network communication and identities in the Taos operating system.

References

1. B. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage and E. G. Sirer, SPIN - An Extensible Microkernel for Application-specific Operating System Services, *Proc. of the 6th SIGOPS European Workshop, ACM SIGOPS*, Wadern, Germany, Sep. 1994, 68-71.
2. D. Engler, M. F. Kaashoek and J. O'Toole, The Operating Systems Kernel as a Secure Programmable Machine, *Proc. of the 6th SIGOPS European Workshop, ACM SIGOPS*, Wadern, Germany, Sep. 1994, 62-67.
3. M. B. Jones, Interposing Agents: Transparently Interposing User Code at the System Interface, *Proc. of the 14th Symp. on Operating System Principles, ACM SIGOPS 27*, 5 (Dec. 1993), 80-93.
4. B. Lampson, M. Abadi, M. Burrows and E. Wobber, Authentication in Distributed Systems: Theory and Practice, *ACM Transactions on Computer Systems 10*, 4 (Nov. 1992), 265-310.
5. *The SPARC Architecture Manual*, Prentice Hall, Englewood Cliffs, NJ, 1992.
6. D. Probert, J. T. Bruno and M. Karaorman, SPACE: A New Approach to Operating System Abstraction, *Proc. of the International Workshop on Object Orientation in Operating Systems, IEEE CS*, Palo Alto, CA., Oct. 1991.
7. S. Radia, M. N. Nelson and M. L. Powell, The Spring Name Service, SMLI Tech. Rep.-93-16, Sun Microsystems Laboratories Inc., Mountain View, CA, Nov. 1993.
8. M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, P. Leonard, S. Langlois and W. Neuhauser, Chorus Distribted Operating System, *USENIX Computing Systems 1* (Oct. 1988), 305-379.
9. A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, J. Jansen and G. van Rossum, Experiences with the Amoeba distributed operating system, *Communications of the ACM 33*, 12 (Dec. 1990), 46-63.
10. L. van Doorn and A. S. Tanenbaum, Using Active Messages to Support Shared Objects, *Proc. of the 6th SIGOPS European Workshop, ACM SIGOPS*, Wadern, Germany, Sep. 1994, 112-116.
11. R. Wahbe, S. Lucco, T. E. Anderson and S. L. Graham, Efficient Software-based Fault Isolation, *Proc. of the 14th Symp. on Operating System Principles, ACM SIGOPS 27*, 5 (Dec. 1993), 203-216.
12. E. Wobber, M. Abadi, M. Burrows and B. Lampson, Authentication in the Taos Operating System, *Proc. of the 14th Symp. on Operating System Principles, ACM SIGOPS 27*, 5 (Dec. 1993).
13. Y. Yu, Automated Proofs of Object Code for a Widely Used Microprocessor, SRC 114, Digital Equipment Corporation, Oct. 1993.