CHAPTER 2 *Background*

By reviewing previous operating system research we attempt to understand the shortcomings of previous architectures and learn from their ideas. This chapter first gives an overview about the main trends in system architecture and then describes several protection mechanisms that have been used in operating systems. Protection is a very central part of the OS and to a largely determines the OS structure. Consequently, most projects that attempted to build extensible and flexible operating systems devote much effort to the protection mechanism. Other important aspects of extensible and flexible operating systems are resource management and modularization.

1 Operating system architectures

Since its early days operating systems struggled for an architecture with a modular structure that increases robustness and allows to adapt the system to new requirements (see Dijkstra [58], Parnas [143], and Haberman [78] for the early discussions about OS modularization). Current systems and research projects follow five architectural roads: monolithic systems with an internal component structure, microkernels with modularization of the system into protected services, object-oriented systems that emphasize ease of development and maintenance, single address-space systems with emphasis on performance and data sharing, and finally language based systems.

1.1 Monolithic systems

Even operating systems that are called monolithic usually have a modular structure and can be extended.

UNIX. Modern UNIX systems can be extended by using so-called modules. The module mechanism is used for device drivers but can also be used to load new file systems. Usually there is a defined interface between the kernel and the module that informs the module when it is activated and deactivated. The module implements functions of a specific function table, such as functions for network devices. This makes the module mechanism very similar to the inheritance mechanism of object-oriented systems. Modules run in the same address space as the system kernel and therefore require the same level of trust as the kernel. They can not be used for untrusted extensions and bugs in extension modules affect the whole system.

Windows NT/2000/XP. Microsoft Windows uses an extension mechanism called Component Object Model (COM). It is an object-oriented interface between components that can be written in different languages. As all components run in one address space there is no protection between them.

1.2 Microkernels

The major drawback of the module mechanisms that is used to extend monolithic systems is the lack of protection. Microkernels attempt to solve this problem be placing components in different address spaces.

Early systems. The concept of a microkernel was developed in the late 1960's. Hansen and colleagues built one of the first microkernels and described its philosophy in an influential paper [81]. The microkernel, called nucleus, makes no assumptions about the kind of applications that run on it. The nucleus provides the process abstraction, client-server-style communication between processes (IPC), creation, control, and removal of processes. IPC between processes is done using message passing. Every process has a message queue and message buffers are maintained in the nucleus in a common pool. Denial of service attacks on this pool are prevented by limiting the number of buffers a process can use simultaneously and by allowing a server to use the received request buffer for sending the reply. The designers of Hydra [190] argue for a separation of policy and mechanism [115]. This influenced other system designers to attempt to build "policy-free" kernels. The external pager mechanism of Mach is one example.

Mach. The Mach microkernel [99] has gained much popularity and is still used today, for example as the foundation of Apples MacOS X/Darwin [8]. Similar to the early microkernels Mach provides a few, hardware-independent abstractions: tasks, threads, memory objects, messages, and ports. Operating system functionality, such as file systems and network protocols runs as a server, called *task*. A task communicates with other tasks and the microkernel by sending a *message* to a *port*[1]. During port communication threads are switched but the timeslice is *handoff* from the client to the server thread. Mach version 3 uses continuations [55] to save memory. A *continuation* is a function that is executed by a re-scheduled thread and a data structure that encapsulates the state of the thread. This data structure can be seen as a reification of the functions execution state. In a synchronous, blocking invocation this state is equivalent to the stack frames.

A system configuration where the complete operating system runs in a single monolithic server is called a single-server system (Mach-UX). Building a single-server system can be accomplished by porting an existing monolithic operating system. Compared to the original monolithic system such a system has the advantage of being able to concurrently run many operating systems, called OS personalities, on the same machine and restarting one OS personality without restarting the whole system. Exchanging a part of the monolithic server is not possible in a single-server system. Therefore multi-server systems (Mach-US) [168] appeared that distribute the functionality of one operating system across many servers, such as tty server, file server, process manager. There is a file-system server, network server, etc. Security is based on ports, which are capabilities. In a multi-server system the communication between OS components is no longer a simple function call but a more expensive IPC. For performance reasons vital parts of the OS server were migrated back into the microkernel. A project at the OSF moved the OSF/1 UNIX server into the kernel [42]. They were able to achieve a performance only 8% slower than a monolithic UNIX. Another example is the network server that is integrated into the NORMA version of the Mach 3 microkernel [14] to achieve fast network IPC.

1. Mach version 2.5 contained a "UNIX compatibility layer" within the microkernel. Release 3.0 [76] moved the UNIX kernel to the application layer.

L4. L4 [117] has a structure that is similar to Mach. It was designed to allow fine-grained servers by providing a fast IPC path. Abstraction by virtualization of the hardware is very important design goal for L4. The Clans&Chiefs [118] model, a custodial access control model [79], is used to validate IPC between protection domains.

Exokernel. While L4 can still be considered a microkernel in the tradition of Mach the Exokernel [61] follows a more radical approach. It assumes that every mechanism embodies policy decisions and thus separation of policy and mechanism is not possible. Therefore the Exokernel attempts to remove also the mechanisms from the kernel and strives for a kernel that only multiplexes hardware between protection domains and provides no higher-level abstractions or virtualization of hardware. Operating systems are deployed as libraries and linked to applications. This is a realization of the idea of an application-specific operating system first proposed by Anderson [174][1]. Engler [60] gives several examples how a library OS can be constructed and resources be safely multiplexed, including network packets and disk blocks. Network processing uses Dynamic Packet Filters (DPF) to dispatch packets to servers. DPFs are written in a simple loop-less language. DPFs are downloaded into the kernel and can be optimized. Disk processing dispatches disk blocks among the servers. To decide which server is allowed to access a certain disk block, the meta data of the file system is reused. As the file system is also implemented as a server the kernel has no information about the structure of the meta data. This problem is solved by using Untrusted Deterministic Functions (UDF). UDFs are written in a turing-complete language and are deterministic. Turing completeness allows the client to use any meta data layout and determinism allows the kernel to use a UDF once with a piece of meta data and when using the same UDF again it gives the same result, thereby guaranteeing that only those blocks are accessed for which access was granted during the initial run of the UDF.

1.3 Single address-space systems

Operating systems that use different address spaces to separate protection domains make it difficult for applications to share data. Single-address-space operating systems (SASOS) attempt to solve this problem by placing all applications in one large (usually 64bit) address space. Applications are protected by using segmentation or page tables with identical page mappings but different page protection. Examples for SASOS are Multics [46], Opal [39], and Mungi [90].

2 Protection mechanisms

An important responsibility of a multi-user operating system is the protection of the data and processes of different users from each other. We will look at five types of protection mechanisms: address spaces, software-fault isolation, proof-carrying code, virtual machines, and type-safe instruction sets.

1. Building operating systems from libraries was common in embedded operating systems and process control systems, such as the IBM System/7. The difference between these systems and Anderson's proposal is the requirement for protection using memory management hardware.

2.1 Address-space based protection

The first operating systems directly used physical memory. With the advent of systems that supported multiple concurrently running processes or tasks memory protection and memory fragmentation became an issue. Fragmentation leads to an under utilization of main memory. Processes could directly access and change the data of other processes, which makes debugging difficult and makes the whole system vulnerable for malicious or erroneous programs. The use of segmentation or virtual address spaces solves these problems. In such a system processes can compute arbitrary addresses but the hardware restricts access to a certain range of addresses and optionally translates addresses from a virtual address space to the physical address space. This allows to better utilize main memory by relocating data and using virtual addresses to access the data. It improves robustness by providing separate address spaces to separate processes. This introduced a new problem: it became difficult to share data. Single address space operating systems (see 1.3) attempt to solve the described sharing problems. They do not use separate address spaces but partition one address space between different protection domains. This is only practical for systems with a large virtual address space, as is provided by a modern 64bit CPU. SASOS continue to rely on MMU protection and continue to be restraint by its limits.

There is excellent support for address-space protection by current hardware. Highly optimized Memory Management Units (MMUs) are part of most processors. Address space protection is course grained and causes a performance overhead during inter-process communication. This motivated the development of other protection mechanisms, such as Software Fault Isolation and Proof-Carrying Code.

2.2 Software Fault Isolation

Software Fault Isolation (SFI) [180] provides memory protection by patching object code. The SFI patcher analyzes the binary code and statically verifies that memory accesses lie within a defined range or inserts instructions that perform this check at run time. SFI was used in VINO [161]) and analyzed in detail in [160]. The major SFI advantage is language independence. The disadvantages are: it must understand the processor instruction set, it uses registers for range checks and it causes runtime overheads.

2.3 Proof-carrying code

Proof-carrying code (PCC) [135] starts from the insight that is much more easier to check the correctness of a proof than to create the proof itself. In the case of PCC the proof states the memory safety of a fragment of code. The proof can be checked and the code can be safely run without address-space protection. PCC can be used to download extensions in a kernel [136].

2.4 Virtual machines

Virtual machines (VM) [126] have the same instruction set as the real hardware but emulate several instructions to provide resource isolation between VMs sharing the same hardware. A control

program [126], also called hypervisor [25], is responsible for the isolation of the VMs. VMs experienced a renewed popularity with the VMWare PC emulator [199]. VMs have two problems: a large granularity and the lack of an efficient inter-VM communication mechanism. The VMWare emulator, for example, consumes a lot of resources to emulate a complete PC which makes it impossible to create fine-grained protection domains. A VM realizes a sandbox. The only interface of this sandbox to the external world are the emulated devices. This makes communication expensive and difficult to control, because the type of the communicated data is lost on the way from the application to the device interface.

2.5 Type-safe instruction set

In contrast to address spaces a type safe instruction set can not only be used to isolate different processes but also to improve the robustness and reliability of the tasks themselves. A type-safe instruction set allows to compute only valid addresses. By making several faults impossible or detecting them at development time a type-safe instruction set reduces the overall number of faults. There are no faults caused by the use of illegal addresses but something similar to faults is still possible. For example, a communication between two protection domains can be aborted because the domain was terminated. The other domain then would experience a fault (exception) during further communication attempts.

What is type-based protection?

Memory protection that relies on type safety must ensure the following minimal system invariants:

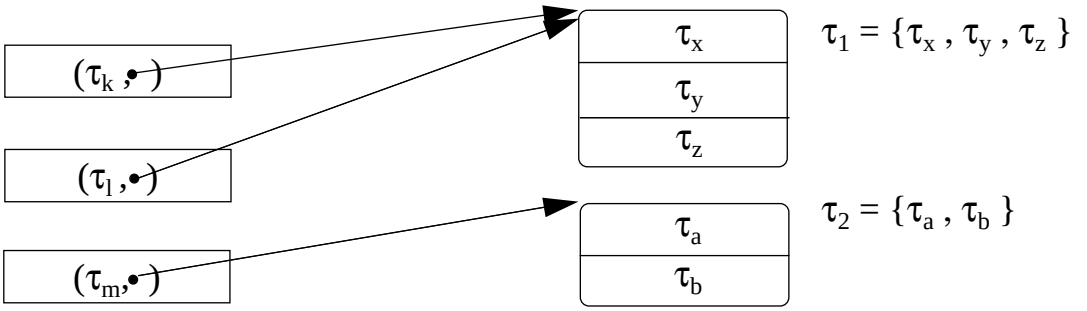
- Every pointer has a pointer type and a value (ptype, value). It is not possible to arbitrarily change the value or type of a pointer. The value of a pointer is changed by assignment.
- Every data entity has a data type (dtype). A data type consists of an ordered set of typed values.
- All instructions that access a data entity use a typed pointer and a numeric or symbolic index into the data entity. That the index is within the boundary of the data entity must be checked either statically or dynamically.

Most type systems have more rules. There are non-pointer data types and type conversions between these types. These rules are intended to avoid programming errors, but are not necessary for protection. Type systems can be classified into systems that check types dynamically whenever a pointer is used or statically at compile time. Static type checking usually is preferable because it does not cause runtime overhead and type errors are detected at development time. Systems that only allow static type checking are usually not expressive enough and therefore many systems use a combination of static and dynamic type checking. Figure 2.1 shows an example of type-based memory protection.

History. Using type safety as a protection started with the language Mesa that first was used on the Alto system [110] about 1973. The Alto supported four languages: BCPL (the ancestor of C), Mesa (a descended from Pascal), Smalltalk, and Lisp. Each language has its own microcoded instruction set. The Mesa language used a space efficient instruction set, also called Mesa [98]. It is an instruction set for a stack machine and is very similar to the Java bytecode instruction set. The development of one of Alto'successor, the Pilot [147], started in 1976. The Pilot exclusively used the Mesa instruction set for protection. As Pilot was a single-user operating systems it was mainly concerned with erroneous code and not malicious code.

Figure 2.1: Type-based memory protection

There are two data entities of type τ_1 and τ_2. Three pointers refer to the data. The first pointer has type τ_k, the second pointer has type τ_l , the third pointer has type τ_m. The system guarantees that pointer types τ_k and τ_l are compatible to data type τ_1 and pointer type τ_m is compatible to data type τ_2.



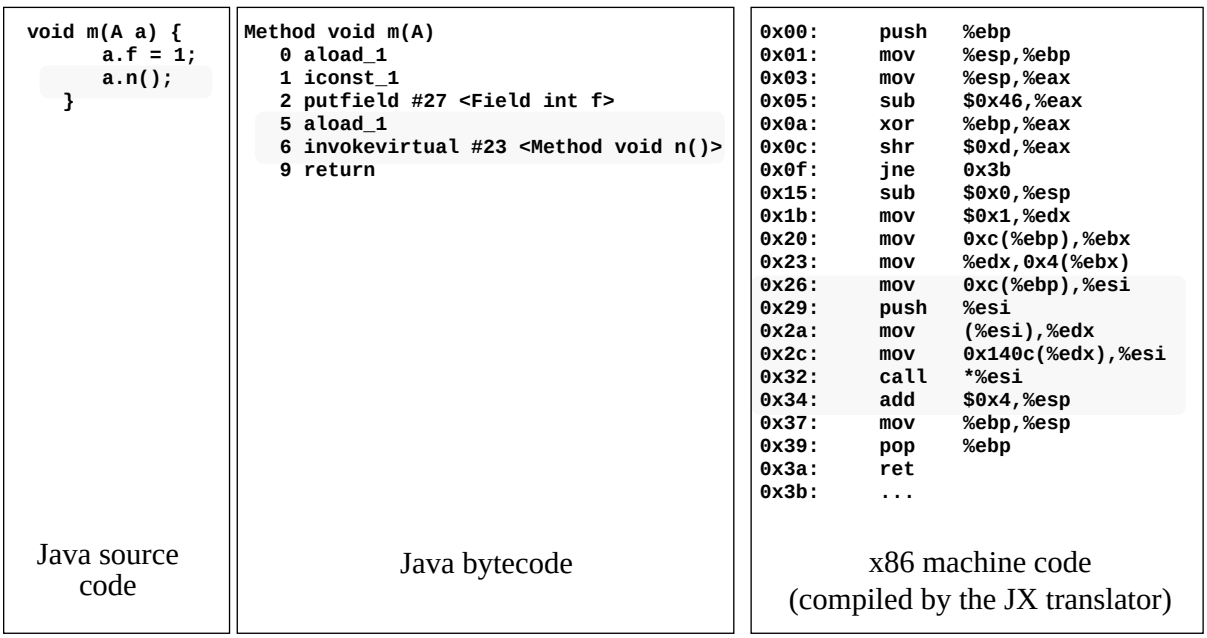
More recent operating systems that use safe languages are SPIN [18], which uses Modula3, and Oberon [189], which uses the Oberon language, a descendant of Modula2.

An operating system which used a type-safe language to achieve high security was KSOS [66], which used the Euclid language [108].

Intermediate instruction set. Using type-safety with off-the-shelf hardware requires the use of an intermediate instruction set. The first intermediate language that was designed but never implemented was UNCOL [170]. The motivation for UNCOL was the diversification of processors and the difficulties of porting machine language programs between these processors. Two of the first widely used intermediate instruction sets are PASCAL's P code [166] and the Smalltalk bytecode [50]. The Inferno [54] system consists of a virtual machine, called Dis, and a language called Limbo. Limbo programs are compiled to the intermediate code of the Dis register machine. While the Limbo language is type safe, the intermediate code is not. Therefore, Inferno relies on a trusted compiler to sign the intermediate code. In the Mahler system [181] the intermediate language was used for code instrumentation and performance studies. The OmniVM system [2] used an intermediate instruction set for mobile code. Most of the intermediate instruction sets are designed for a specific high-level language. This often leads to inefficient code when compiling other languages to the intermediate instruction set. Typed Assembly Language (TAL) [132] is an attempt to create a low-level type-safe language that can be used as the target instruction set for many languages and efficiently compiled to a real CPU instruction set. Microsofts .NET architecture also relies on an intermediate instruction set, called Microsoft Intermediate Language (MSIL) [59], which is a type-safe instruction set. MSIL was designed as the target instruction set for different languages, such as C# and Visual Basic. The JX system uses Java bytecode. Java bytecode is an instruction set for the Java Virtual Machine (JVM) which is an abstract stack machine. Figure 2.2 shows how the Java source code language is translated to the bytecode instruction set and then to the instruction set of the real CPU, which is a x86. The instruction set of the JVM uses static typing with few dynamic type checks. There are two kinds of types: pointer types and numeric types. There are two data areas, the heap and the stack, and three

Figure 2.2: Relation between Java source code, Java bytecode, and machine code

A type-safe Java source program is compiled to a type-safe intermediate instruction set. This instruction set can be interpreted or compiled to the instruction set of the real processor. The shaded area shows the representation of a method invocation in Java source, bytecode, and machine code.



classes of instructions that must preserve typing of data: instructions that load a word from the heap and store it on the stack or vice versa and instructions that call methods that expect parameters on the stack and push the return value. From a memory-protection point of view type-safety must ensure that there is no data flow between memory cells (heap or stack) that store numeric types and memory cells that store pointer types. When the same memory cell is used to store both kinds of types, as are stack cells, it must be assured that they are re-initialized before being used with a new type. When data flows between cells with different pointer types (pointer assignment) the object-oriented subtyping rules apply. Pointers can be assigned from subtype to supertype but not in the opposite direction. When this is necessary a dynamic type check must be executed using the checkcast instruction. The bytecode verifier ensures that this instruction is present in the appropriate place.

Assuring type safety. Building an operating system that completely relies on the type safety of the JVM instruction set requires a high confidence in the correctness of the JVM specification with respect to type safety and the proper operation of the bytecode verifier. There have been many attempts to formalize the JVM specification (see for example [145]). These projects uncovered and fixed errors in the following areas of verification: ensuring that an object is initialized before it is used [150], typing of subroutines [167], dynamic class loading [153].

3 Resource protection and management mechanisms

In addition to protect the memory of tasks an operating system has to manage and protect resources. Tasks and untrusted kernel extensions consume resources and modify the state of a trusted component. Approaches to handle this are hierarchical resource management, resource containers, transactions, and path-based resource management.

3.1 Hierarchical resource management

The first microkernels also introduced a hierarchical resource management (HRM) scheme, called nested processes [81]. The process hierarchy, which is usually defined by the parent/child relationship, is used to manage and account for resources. A child obtains resources from its parent and these resources are deducted from the parent. HRM is used in Fluke and Alta [177]. ANTS decided against a hierarchical model because they consider it to be too restrictive [107]. The argument is that the resource permissions of an active code domain should be determined by its credentials and not by the permissions of the ANTS execution environment that creates the domain.

3.2 Resource containers

The purpose of resource containers [13] is to separate the two abstractions protection domain and resource principal and account for resources that are consumed inside the kernel on behalf of a process. Although Banga et al. only implemented resource containers to control CPU usage, they state that the resource container abstraction is more general and can as well be used for the management of other resources, such as disk bandwidth and TCP buffers. A resource principal can even be more fine grained than a process. As their major example is a web server it is beneficial to account resources per connection or per thread that handles a connection. This is of course only possible if the thread does not behave maliciously and only uses defined interfaces to access resources within its own process.

3.3 Lightweight transactions

Transactions are used in VINO [156] to prevent resource misuse of kernel extensions (grafts). VINO transactions are designed to be lightweight. Nevertheless the cost of transaction management may outweighed the benefits of the kernel extension.

3.4 Path-based resource management

Scout [134] uses the abstraction of a path for resource management. Resources are not accounted to a process, protection domain, or module, but to the execution path of an activity. This execution path can cross module boundaries. The Escort architecture [164] ensures that all dataflow in Scout is represented as a path. This is essential to guarantee security and the confinement of information. The Scout team ported a JVM to the Scout system (Joust [83]) and applied path-based resource control to Java programs.

3.5 User-defined resource management

A study of Mach performance measured that unnecessary multiprocessor locking (on a uniprocessor) was responsible for about 10 percent of the overhead of running the OSF/1 system on top of Mach compared with the OSF/1 integrated kernel [42]. This and the fact that a large number of all systems are uniprocessors make it worthwhile to allow application specific scheduling to avoid heavy-weight locking using mutual exclusion locks.

4 Modularity and reusability

A modular design and reusability have been important topics since the early days of operating systems. Habermann [78] proposed a layered design called "functional hierarchy", where each layer provides a new "virtual machine" abstraction to the upper layers. Parnas [143] and Dijkstra's "THE" system [58] also emphasize modularity.

Depending on the mechanism that is used to encapsulate modules a different communication mechanism is needed: procedure call or message passing. Lauer & Needham [113] observed that these are equivalent abstractions that even can be mapped to each other at a conceptual level.

Modularity is especially important for embedded systems. These systems have severe resource constraints and therefore cannot accept unneeded functionality. Furthermore they often must be customized to a specific application domain. Therefore it is not astonishing that many projects that emphasize modularity have an embedded system background, for example, MMLite [91], Pure [20], Pebble [71], and 2K [105]. Modularity and reusability is the main theme of the following systems:

x-Kernel. The x-kernel [97] is an operating system that focuses on the flexible construction of communication protocols. It allows to compose new protocols by using existing components. Everything in the x-kernel, including devices, is represented as a protocol and the system has strong supports for the composition of protocols. The x-kernel demonstrated that this flexibility does not come with a performance cost. The x-kernel protocols are faster than equivalent UNIX implementations.

Lipto. Lipto [57] is an OS derived from the x-kernel. It considers modularity and protection as two orthogonal concepts. Lipto uses proxies [158] to uniformly refer to local and remote objects. It uses a proxy compiler to generate C/C++ code from an interface definition. This complicates its use from a programmers perspective and restricts proxies to classes that are specially prepared at development time.

Microsoft COM. The Component Object Model (COM) [26] is the component model of Microsoft's Windows operating systems. COM is a language-neutral, binary interface standard and an architecture that defines an execution environment for components. An important part of this architecture are so called apartments. The main purpose of apartments are to mix components that were written in a single-threaded style (without synchronizing access to the objects), called single-threaded apartments (STAs) and components that use multiple threads, called multithreaded apartments (MTAs). As COM was introduced 1993, at a time when Windows did not support threads, there was a lot of code written in a single-threaded style. The two types of apartments can be mixed in a single process. Calls between incompatible apartments are serialized by using a proxy instead of a direct object reference.

The programmer must be careful not to share a direct reference to an object between different incompatible apartments, for example by using a global variable. Although COM provides the ability to run a component in a separate process (out-of-process, EXE server), communication with such a component is orders-of-magnitude slower.

OSKit. The OSKit [67] attempts to leverage the huge amount of UNIX-like open source operating system code to build new operating systems. The kit provides glue code to use code from different systems in a single system, like a NetBSD network stack with a Linux file system. To achieve this reusability of independent components, existing systems are componentized and equipped with a COM interface.

MMLite. MMLite [91] is an architecture to build a complete OS out of object-based components. It is similar to the OSKit approach but allows to even replace the scheduler component or the virtual memory management component. COM is used as an interface between components, which usually are written in C or C++. Because modularity is at the object level, the memory of objects can not be deallocated explicitly. A reference counting garbage collector is used for the heap management, which has well-known limitations, such as the inability to collect cycles and an overhead for the maintenance of the reference counts. Address-space protection is used to provide firewalls but is otherwise orthogonal to the modular decomposition of the system. Its conceptual architecture is very similar to Lipto.

5 Extensibility and openness

As many of the previously presented concepts the idea of an open operating system is fairly old [109]. There are two categories of openness: systems that allow a gradual modification using an extension interface and systems that allow the wholesale replacement of components. The first category includes Mach external pagers [191], extensible file systems [122], as well as applications that are extensible by providing a so-called plug-in interface. Examples are QuarkXpress, Photoshop, Netscape, and Gimp. The extensions are loaded in the address space of the application and therefore must be as trustworthy as the application itself. Current UNIX kernels can be extended by using modules. As they run in the kernel address space, they must be trusted like the kernel. This situation motivated many projects to support untrusted extension modules. The most prominent of them are SPIN and VINO.

SPIN. SPIN [18] is an operating system that can be extended by using so-called event handlers. Event handlers are portions of untrusted code that can be downloaded into the kernel. Kernel operations trigger events [142] that can be processed by a handler. Memory protection is realized using a type-safe language to write handlers (Modula-3) and a trusted compiler that signs the handlers. Whether a handler is allowed to handle an event is decided by guards. If the handler runs longer than a certain time limit it can be terminated. Only handlers that are marked as EPHEMERAL can be terminated. Such a handler can only call other functions that are marked EPHEMERAL. No kernel function is marked ephemeral and therefore it is safe to abort an ephemeral handler.

Handlers are downloaded in the kernel as a SPINDLE. A system service in SPIN is decomposed into a SPINDLE, a library, and a user-level server. A SPINDLE can be downloaded into the supervisor address space, a library is loaded into the application address space and can contact the

SPINDLE via system calls, and the user-level server is used to maintain the state of the service that outlives the application.

VINO. VINO uses SFI for memory protection. (MiSFIT [161], the "Minimal i386 Software Fault Isolation Tool"). Extensions are called grafts. Resource consumption of grafts is monitored and an extension module can be aborted like a transaction. Running grafts in the kernel is usually motivated by performance concerns. The overhead of the transaction mechanism may render the extensions useless for the performance goal.

6 Summary

Current OS architectures are not able to cope with demands of today's applications. We attribute most of the shortcomings to the used protection mechanism: the address space. Other protection mechanisms are better suited because they offer more flexibility and finer granularity. In this dissertation we develop an operating system structure using a type-safe instruction set as the only protection mechanism. Other protection mechanisms that use compiler extensions force the system to use a trusted compiler that signs the compiled code.We learn from Java that a general purpose language can be compiled to a type-safe instruction set. All of the extensible systems (SPIN, VINO, Exokernel) use a combination of several memory protection mechanisms. They use address-space protection in combination with type safety or Software Fault Isolation. We think that this unnecessarily complicates the architecture and advocate an architecture that relies on only one protection mechanism: type safety.

Using a different protection mechanism allows a new view at system problems of the past and reconsider if and how their solutions can be transferred to our new architecture. We learn from the Mach project to structure the system as a multi-server system to allow restartability, extensibility, and fault containment. We should not force components to use IPC if not necessary. We learn from SPIN that type-safe protection allows fine grained extensibility. Microkernels like Mach and especially the Exokernel emphasize that resources should be managed outside the kernel. Combining this with a type-safe protection system is a challenging task as such systems require support by a runtime system that contains many resource management mechanisms and policies.

Virtual machines, or hypervisors, are able to isolate systems running on the same hardware, but they have no support for sharing and efficient, typed communication.

Microsoft COM apartments demonstrate, that the discussed problems are not only of academic interest. Apartments subdivide a process in different execution environments. Objects must not be shared between incompatible apartments, i.e., apartments that use incompatible thread scheduling strategies. COM can not *enforce* the isolation between apartments, because a programmer error may lead to direct sharing of objects, for example when using a global pointer to access the same object in two or more apartments. A malicious component can not be isolated at all. This shows that a protection domain should not be separated from execution environment but instead the overhead associated with a protection domain should be reduced.

Previous operating systems that used type safety for protection were designed as single-user systems (Pilot, JavaOS) or used type-safety merely as an additional protection mechanism inside one address space and used address spaces to isolate the tasks of different users (SPIN). Using two pro-

tection mechanisms complicates the system architecture and application development and also requires two different resource protection mechanisms. We will show that type safety is sufficient as a protection mechanism to build a multi-user operating system.