

# Issues in Extensible Operating Systems

Margo I. Seltzer, Yasuhiro Endo, Christopher Small, Keith A. Smith

## 1. Abstract

Operating systems research has traditionally consisted of adding functionality to the operating system or inventing and evaluating new methods for performing functions. Regardless of the research goal, the single constant has been that the size and complexity of operating systems increase over time. As a result, operating systems are usually the single most complex piece of software in a computer system, containing hundreds of thousands, if not millions, of lines of code. Today's operating system research is directed at finding new ways to structure the operating system in order to increase its flexibility, allowing it to adapt to changes in the application set it must support. This paper discusses the issues involved in designing such extensible systems and the array of choices facing the operating system designer. We present a framework for describing extensible operating systems and then relate current operating systems to this framework.

## 2. Introduction

Approximately every ten years the operating systems community undergoes a re-engineering effort shifting the focus of operating system research. In the 1950's, operating systems were batch monitors, automating the process of setting up users' jobs. The operating system was little more than a set of libraries used to load, compile, and execute users' programs. Time-sharing operating systems appeared in the 1960's, allowing the computer to run more than one job at a time. The 1970's can be characterized by the emergence of the portable operating system (e.g. UNIX [17]), initially a small, simple system that has, over time, become yet another large, complex system. The 1980's heralded three new technologies: the micro-kernel architecture [1], in rebellion against the increased complexity of 1970's systems; personal computer operating systems (e.g. MS-DOS); and network-aware operating systems (e.g. Sprite [16]). As the line between personal computers and workstations blurs, so does the line between personal computer operating systems and other operating systems. The resulting systems are large and complex, leading to the research thrust of the 1990's: extensible operating systems.

No single person, or even small group of people, can understand an entire complex operating system; it has become ever more difficult to extend or adapt systems to support new applications that make unanticipated demands on the operating system. In order to reach solutions, a new set of design decisions and research questions arises. In this paper, we discuss the key issues that must be addressed in designing an extensible operating system. For each issue, we present the solutions under consideration today. Finally, we conclude with a brief survey of today's systems, discussing how they have addressed each of the issues presented.

### 3. Why Extensibility

The move towards extensibility is motivated primarily by a need for better performance and more functionality. Moving application code into the operating system can reduce the number of context switches, thereby improving performance. Additionally, extensions can be used to create data paths in the kernel that otherwise would not exist. For example, in a traditional system an application that moves data from the network to a local disk copies the data from the network into the application and back into the operating system to the file system. This results in moving potentially massive amounts of data across the user/kernel boundary several times for no good reason. While special purpose solutions to this problem have been implemented [8], a general approach is to allow applications to download a kernel extension that creates a path directly from the network to the disk, removing the need for extra copies [9].

Accommodating unforeseen or special-purpose functionality provides a second motivation for extensibility. There has always been a class of applications that are not well-served by conventional operating systems (e.g. database management systems, real-time applications). In most cases, these applications go to great lengths to avoid using the services provided by the operating system or they use a special purpose system, optimized for the specific application. In addition, as new applications emerge, they undoubtedly require new functionality from the operating system. As this new functionality is added to an operating system, it becomes less reliable and too large to maintain and adapt effectively. The vendors of personal computer operating systems identified the need for extensibility years ago, and as a result, a system without much power or functionality (i.e. MS-DOS) has survived and thrived in the face of technically superior solutions.

MS-DOS is often ignored by the research community, but two important features differentiate it from other systems. First, it is the single most widely used operating system in history. Second, MS-DOS has minimal native functionality, but possesses the ability to extend its functionality through system calls and device interrupt hooks. For example, the MS-DOS kernel does not provide disk caching for its file system, but caching is often provided by an extension, such as SMARTDRV disk caching software [13], that sits between applications and the standard disk device driver.

The MS-DOS model of extensibility is not without its problems, however. Since MS-DOS implements a single address space, a fatal bug in any program or extension crashes the entire system. Additionally, since these extensions stay resident, they reduce the effective memory for other applications. The MS-DOS extension mechanism adds a level of redirection on every system call, substantially lengthening the code path executed [5].

Researchers in extensible operating systems are trying to solve problems that plague real systems. One challenge is to find a way to provide the flexibility of MS-DOS, while also providing high performance and security.

Research on these same problems has come in different guises. For example, the goal of the Java project, originally from Sun Microsystems Laboratories, is to develop a safe language and runtime system for building distributed extensible applications. The Java language has neither

untyped pointers nor explicit memory allocation and deallocation, and array indices are bounds-checked at run-time. This simplifies the task of creating stable, extensible systems.

The *reflective systems* community offers a model of computation in which systems have two types of interfaces, the computational interface, which invokes a service, and the meta-interface, which controls how the system works. In the context of operating systems, one can consider the system calls the computational interface and the extensibility mechanism the meta-interface. The closely related *Open Implementation* software design technique [12] allows the clients of a module to influence the module's implementation strategy. For example, an open implementation sorting module would provide an interface to sort data (the computational interface) and a separate interface to select a particular sorting algorithm (the meta-interface).

#### **4. Design Points in Extensible Systems**

The main challenge in building extensible systems is to provide applications with maximum flexibility without exposing the applications to unreasonable risk. The following aspects of this problem are discussed below: *Mutability and Location*, the measure of a system's flexibility and where it is provided; *Trust and Failure*, the effects of errant extensions; *Lifetime*, the time during which extensions remain in effect; *Granularity*, the unit of extensibility; and *Arbitration*, how the system copes with conflicting extensions. Table 1 summarizes the design points under discussion, presenting the options associated with each design point.

##### ***Mutability and Location***

A conventional kernel is a single executable with its own address space. To modify the kernel, its source code is updated and the system is recompiled and relinked to form a new executable. The modifications take effect when the system is rebooted with the new version. In a multi-user system, revising or replacing the operating system requires privileges not typically granted to ordinary users or applications. In general, in a conventional system, applications have no choice but to use exactly those services provided by the original operating system.

In making a system extensible, there are three questions to answer: is the operating system code base fixed at kernel build time or can it be extended at run time; is the behavior of a running system static, or can it be changed dynamically; and are extensions implemented at user-level or kernel-level. The first two questions determine the *mutability* of the system while the third indicates the *location* of extensions, implying the types of protection mechanisms that might be employed to protect the kernel from errant extensions. Table 2 shows the cross-product of mutability options for a system, providing examples of systems using each approach.

It is important to note that a single system might employ multiple approaches to extensibility and can implement its approach either at user-level or at kernel-level. For example, the Solaris Operating System provides support both for loadable device drivers (an extensible mechanism since it permits new code to be added, thereby changing the behavior of the system) and for scheduling algorithm selection on a per process basis [21] (a reconfigurable mechanism as the scheduling policies are fixed at system build time, but may be selected dynamically at run time). In the case of Solaris, both of these mechanisms are implemented at kernel-level. In contrast,

<b>Design Issue</b>	<b>Options</b>		
<b>Mutability</b>	<b>parameterized</b> both the code base and configuration are determined at compile time	<b>reconfigurable</b> the code base is determined at compile time, but behavior can be modified at run time	<b>extensible</b> both the code base and behavior can be modified at run time.
<b>Location</b>	<b>user</b> extensions are implemented at user-level	<b>kernel</b> extensions are implemented at kernel-level	
<b>Trust/Failure</b>	<b>good will</b> extensions can read/write any portion of the kernel	<b>software</b> safe languages or SFI protect kernel from extensions	<b>hardware</b> address spaces protect kernel from extensions
<b>Lifetime</b>	<b>application</b> extensions persist as long as application runs	<b>resource</b> extensions persists as long as the resources to which they are attached	<b>kernel</b> extensions persist until kernel reboot when reinitialization takes place
<b>Granularity</b>	<b>module</b> entire subsystems are replaced	<b>limited procedural</b> a subset of the kernel functions can be replaced	<b>procedural</b> any function can be replaced
<b>Arbitration</b>	<b>prohibition</b> request fails if there is a conflict (e.g. there is a central authority that arbitrates requests)	<b>split resolution</b> conflicts are avoided by leaving global decisions to the kernel and local decisions to application	<b>multiplexing</b> resources are time multiplexed

**Table 1. Extensible Operating System Taxonomy.**

		<b>Code Base</b>	
		<b>fixed</b>	<b>extensible</b>
<b>Configuration</b>	<b>static</b>	Code and behavior fixed at compile time. System: Scout Name: <b>parameterizable</b>	<b>Not feasible: this choice implies an extensible code base with static behavior.</b>
	<b>dynamic</b>	Code is fixed, but the behavior can be modified at run time. System: Solaris (scheduling) Name: <b>reconfigurable</b>	Both code and behavior can be modified at runtime. System: SPIN, VINO Name: <b>extensible</b>

**Table 2. Kernel Mutability.**

microkernels such as Mach, permit extensibility through the addition or modification of user-level servers. Since such servers extend the code base and modify the behavior of the system, they are classified as user-level extensible systems.

It is useful to view the different categories of mutability as incrementally adding flexibility to conventional systems. The first incremental step from a conventional kernel to an extensible one is to provide compile-time options that generate operating systems with different behaviors. These application-specific or *parameterizable* operating systems are based on the notion that most systems run a single class of application, so the operating system should be tailored to the specific workload running on a particular machine. To this end, the kernel is a toolkit from which the appropriate modules are drawn to create a specialized operating system [15]. The resulting system is tailored to a class of applications' needs, but is not dynamically adaptable.

The next incremental step in making a system dynamically adaptable is to modify the behavior of a fixed code base at run time. As mentioned earlier, the scheduling algorithms in Solaris are fixed at system build time, but the scheduling algorithm used by a particular process can be modified at run time. This is the hallmark of the *reconfigurable* system.

Finally, we reach the *extensible* systems, which permit new code to be incorporated into the system at run time. In microkernel systems, such as Mach [1], the new code is encapsulated as user-level server processes. Alternatively, in newer designs, such as the Exokernel [7], all operating system services are moved to user-level libraries leaving only simple hardware abstractions in the hardware. Applications select the kernel services they want by selecting appropriate libraries, replacing any default behavior simply by providing a new implementation of a kernel service.

Instead of moving kernel functionality into user-space, an alternative approach is to move application functionality into the kernel. In systems using this approach, applications download code into the operating system in order to alter its behavior. MS-DOS falls into this category, as it allows arbitrary applications to change its behavior. New systems that provide better protection as well as extensibility use safe languages (e.g. Modula-3) or other software fault isolation techniques (e.g. *sandboxing* [22]) to limit the ability of an extension to read or modify the rest of the kernel. These systems allow applications to directly modify the kernel avoiding costly cross-address space calls. However, when applications download code, the safety issues discussed below become critical. Furthermore, the precise method of downloading code can limit the types of modifications that an application may make. This limitation can be either beneficial or detrimental.

One way to implement a downloadable system is to write extensions in an interpreted language and embed an interpreter in the operating system. The interpreter can enforce whatever safety constraints are desired, but there is a runtime cost associated with interpreted languages. This approach is used in application-specific cases such as network packet classifiers [14].

In each of the descriptions above, we have assumed that applications take explicit action to extend the system. It is also possible to have the system adapt to application needs, using specialized code paths to provide improved functionality or performance. For example, in the Synthetix system [6], application needs can be inferred through hints provided by the application via meta-

interfaces [11] or simply by examining the system call behavior of applications. This approach, called *inferred adaptability*, is orthogonal to the particular extension technology employed by a system. Once the kernel has determined a need to adapt, it can effect the adaptation by selecting an existing, but not necessarily default, code path in a reconfigurable system. Alternately, it could trigger new code to be added to the running kernel in an extensible system.

### ***Trust and Failure***

Different extension technologies imply different levels of trust between the applications and the operating system. A fundamental question is how much power extensions have, that is, how much they are trusted not to corrupt the rest of the system, and when they fail, how much of the system will be affected by the failure. There are three ways of protecting the kernel from ill-behaved extensions: *good will*, *hardware*, and *software*.

MS-DOS has relied on the good will approach: any application can add extensions to the operating system, and if those extensions are faulty, the entire system crashes.

Traditional systems and microkernels use hardware protection: the kernel runs in its own address space and only the kernel is allowed to read or write its data structures. In a hardware protected model (e.g. a microkernel), if an extension fails, only the process implementing the extension fails. However, if most applications are using the service, failure of the service is tantamount to system failure. For example, if a file system server crashes, it is unlikely that any application using that service can make forward progress.

There are a variety of mechanisms for providing software protection. If extensions are written in a safe language (e.g. Modula-3), then extensions can be prohibited from accessing parts of the system to which they have not been granted access. Safe languages can be compiled or interpreted. Another approach is to take code written in any language and then make it safe by incorporating runtime checks. In each of these safety models, faults in an extension are isolated to the applications using the extension. Both approaches prevent a single kernel extension from jeopardizing the integrity of the entire system, so there is no objective measure of which software approach is best. Given that they are all functionally equivalent, one might consider the performance impact of each approach when selecting a technology [19].

In both the hardware and software protection approaches, the system must protect itself against a malicious extension monopolizing resources. This form of protection is easy for hardware schemes, since the same mechanisms that protect the kernel from misbehaving applications can be used to protect the kernel against misbehaving user-level extensions. In the software schemes, protection is far more challenging. In addition to using software to prevent illegal data access (e.g. wild writes or reads to disallowed memory locations), the interfaces available to an extension must be carefully designed to prevent extensions from obtaining or disseminating information that would violate the security of the system. Additionally, extensions must be preemptible to prevent a runaway extension from monopolizing the processor. More generally, extensions must be prohibited from holding kernel locks or limited kernel resources (normally not available to user level processes) for extended periods of time. The particular mechanisms used to enforce these constraints vary with the particular software technology used.

## *Lifetimes*

The previous two sections have discussed extensions as if they were associated with a particular application. It is instructive to consider explicitly to what object extensions are bound and how long they persist. If an extension is associated with a specific application, then the extension's life time is tied to that of the application. As long as the application is running, the extension can remain in effect; if the application is not running, the extension is not in effect. When the application is modified or recompiled, the extension can be as well. However, there are cases where extensions should be associated with objects other than applications. For example, consider an extension that encrypts and decrypts files when they are written to or read from secondary storage. Such an extension must remain in effect as long as any files that were written using it exist. This is a particularly stringent requirement in that files persist across applications, system reboots, and system upgrades. In theory, it is desirable for extensions to persist as long as the resource to which the extension is bound persists.

In some cases, the lifetime of a resource, such as a file, can be measured in years. *Watchdogs* [2] are a file system feature that associates extensions with long-lived resources. In the prototype implementation, a UNIX file system was modified to attach a program (the watchdog) to a file. When an application attempted to perform an operation on the file, the request was sent to the watchdog program, which decided how to respond. Watchdogs were used to implement compressed or encrypted files, or to add a layer of file access control on top of that provided by UNIX.

DOS provides other examples of long-lived extensions. For example, the only file system included in the DOS kernel is the FAT file system. Additional file systems, such as the compressed file system, DRVSPACE, and the CDROM file system can be added as extensions at boot time. If the extensions are not added and the user tries to access a file, originally stored in the file system (e.g. a compressed file), the data returned are garbage. In the case of the CDROM access, failure to install the file system prevents the user from seeing any of the names or data stored in that file system.

In the rest of this paper, we categorize extension lifetimes in terms of the object to which the extension is bound. *Application lifetime* means that the extension persists only as long as the particular application(s) using it persist. *Resource lifetime* means that the extension persists as long as some particular resource exists; watchdogs [2] are an example of a resource lifetime extension. *Kernel lifetime* means that the extension persists as long as the kernel is running, but that some form of explicit configuration is necessary upon reboot (e.g. installing the compressed file system). *Permanent* means that once an extension is installed, it remains in effect until explicitly removed; an extension statically compiled into the kernel falls in this class.

## *Extension Granularity*

The question of extension granularity is related to extension lifetime: what is the unit of modification for an extension? The *granularity* of an extension is a function of the kernel interface provided to the extensions (*internal interface*) and the interface through which applications access extensions (*access interface*). The internal interface dictates how self-

contained extensions must be, while the access interface dictates the type of functionality that is reasonable to implement in an extension.

Internal interfaces can permit access to: application-available interfaces (e.g. system calls), a subset of internal kernel interfaces, or all kernel interfaces. As the number of exposed kernel entry points increases, the extensions become more closely integrated with the operating system. This reduces the size and complexity of extensions, but demands more sophistication on the part of the extension designer, increases the potential for extensions to become incompatible with future versions of the kernel, and increases the overhead of guaranteeing protection against misbehaving extensions.

There are three general approaches to modifying the access interface: replacing or overloading existing kernel entry points (the MS-DOS approach), replacing or overloading internal kernel entry points, or using general purpose messages to communicate with an extension.

In MS-DOS, both the access and internal interfaces are identical. Extensions are accessed by augmenting the system call or BIOS handler. Similarly, the only entry points from an extension back to MS-DOS are through system calls and BIOS calls. We call this coarse grain extensibility *module extensibility* because each extension is self-contained, using only the kernel interfaces already available to applications.

A conventional microkernel is another example of a module extensible system. In the microkernel architecture, an entire server can be replaced, but this server must be a self-contained unit, using only the calls available to other user-level processes.

Finer grain extensibility is supported by providing a mechanism whereby an extension can override any internal entry point. For example, when the operating system is implemented as a user-level application library (e.g. Exokernel), an application can replace any single kernel module with one of its own. Therefore, it is possible to replace both the internal and external interfaces with application-specific ones. We call this architecture *procedural extensibility* because extensions can be defined in arbitrarily small units.

When a system places extensions in the kernel and uses software protection, procedural extensibility is often impractical, since it implies that every routine in the kernel must be protected. Any assumptions the kernel makes before or after calling a procedure are invalidated if any procedure can be replaced. As a result, the software protected systems typically use a special case of procedural extensibility, *limited procedural extensibility*, in which only a subset of kernel functions may be extended. In these systems, the challenge is to identify all the useful extension points and ensure that their replacement cannot invalidate system assumptions.

### ***Arbitration Between Conflicting Extensions***

The final issue we consider is that of arbitrating between conflicting kernel extensions. Once applications are given the power to affect kernel policy, it is likely that competing applications will make conflicting requests. Extensions can conflict in three ways. They can conflict because they require the same physical resources (e.g. in MS-DOS two extensions could require the same physical address). We call this *interference*. They can conflict because each is demanding a

fraction of a resource and the sums of the requests is greater than one (e.g. memory, processor cycles, network bandwidth). We call this *resource contention*. They can conflict because of differing policy decisions (e.g. fair share scheduling versus round-robin scheduling). We call this *policy contention*.

Resource requests can be *hard* or *soft*. A hard request is one that must be fulfilled for an application to proceed; if the resource is not available, the application has no recourse but to terminate. A soft request is one where the application may be able to proceed with a partial allocation of the resource. In many cases, for example, if an application can not acquire physical pages for all of the virtual memory it allocates, it is acceptable to allow some of its memory to be paged to disk; the request for memory is soft. On the other hand, a video conferencing application may require a minimum fixed bandwidth in order to maintain acceptable image quality and frame rate; the request for bandwidth would be a hard request.

There are three basic approaches to dealing with conflict: disallow conflicts, make conflict impossible by dividing resource allocation between a global allocator and local allocators, and multiplex resources. The simplest approach is to disallow any conflicts. This seems to be the only reasonable approach when the conflict is over a physical resource that cannot be multiplexed or when it is a hard resource request. We call this *prohibition*.

We can also remove conflict by decomposing the decision process into global decisions and local decisions. The kernel makes global decisions, allocating resources to processes or groups of cooperating processes. Then, once allocated, the processes with the resources make local allocation decisions. Consider two sets of applications, one which should be scheduled round-robin, and the other which should be scheduled based upon the outstanding tasks to be processed. If these are hard requests, then one request cannot be granted; the conflict is disallowed. Alternately, the kernel can employ its own scheduling discipline to select one of the two process sets, and then allow each set to use its desired algorithm to schedule the processes that comprise the set. We call this partitioning of responsibility *split resolution*. Split resolution is appropriate for soft resource contention and for policy contention but does nothing for interference.

The only solution for interference, other than prohibition, is *multiplexing*. For example, if two extensions require the same physical address, but only one extension needs the address at any particular time, it may be possible to time multiplex the use of the resource. In the case of virtual memory requests, multiplexing is accomplished through paging or swapping. If multiplexing cannot be supported and an extension encounters interference, some extension must fail.

For the first four issues we presented, a system could select one of the possible alternatives, but in the case of arbitration, we believe that any extensible system will have to support all three models.

## 5. Example Extensible Operating Systems

In this section we discuss several examples of the extensible systems under development today, focusing on the decisions made concerning the issues raised in this article and summarized in Table 1. This section is by no means exhaustive. We have selected a range of systems that portray different points in the design space and that represent different operating system communities,

such as the commercial sector, the main stream operating system research community, and the reflective community. The systems are summarized in Table 3.

System	Mutability	Location	Trust/Failure	Lifetime	Granularity
<b>Apertos</b>	extensible	kernel	software	application	procedural
<b>Exokernel</b>	extensible	user	hardware	permanent	procedural
	extensible	kernel	software	application	limited procedural
<b>Mach</b>	extensible	user	hardware	kernel	module
<b>MS-DOS</b>	extensible	kernel	good will	kernel	module
<b>μChoices</b>	extensible	kernel	software	kernel	module
<b>Scout</b>	parameterizable	kernel	good will	permanent	procedural
<b>SPIN</b>	extensible	kernel	software	kernel	limited procedural
<b>Solaris</b>	parameterizable	kernel	good will	permanent	module
	reconfigurable	kernel	good will	application	module
	extensible	kernel	good will	kernel	module
<b>VINO</b>	extensible	kernel	software	resource	limited procedural

**Table 3. Example Extensible Operating Systems.** The Exokernel and Solaris systems support multiple technologies for extensibility; each technology is shown independently. Most extensibility in Exokernel is achieved by modifying user-level libraries (the first entry), but Exokernel also provides the ability to download code into the kernel (the second entry). Solaris supports build time parameterization in the form of the addition of new file systems or scheduling classes (the first entry). At run-time, the super-user can dynamically assign processes to different scheduling classes (the second entry) and can dynamically download device drivers (the third entry).

### *Apertos*

Apertos [10] is designed to support an open and mobile computing environment, as well as to introduce a new object-oriented framework to combat complexity in large systems. Apertos consists solely of objects that operate in a heterogeneous environment. The system is composed of objects that contain information and meta-objects that define the semantics and behavior of objects. An Apertos system changes dynamically by permitting objects to freely associate and dissociate themselves with collections of meta-objects (called *meta-spaces*). In addition to supporting the reconfiguration of objects via association with different meta-spaces, Apertos provides extensibility through the downloading of protected code. This downloading capability makes Apertos an extensible system in which software-protected extensions reside at kernel-level. Since extensions and behavior modifications occur in terms of object methods, its granularity model is procedural, and its extensions persist as long as the reconfigured objects exist, so we consider it to have application lifetime.

## *Exokernel*

The goal of the Exokernel project [7] is to remove abstractions from the kernel and export a low-level machine interface directly to applications. User-level libraries implement most of the abstractions traditionally implemented by the kernel, and the kernel implements the bare-minimum functionality required to export the hardware interface to applications safely.

There are two ways to extend the Exokernel. The first is to modify the user-level libraries that implement the kernel abstractions. This model is procedural extensible; because abstractions are user-level code, they can be modified and extended to an arbitrary degree. Since these extensions run at user-level, the kernel is protected from misbehaving extensions by hardware protection. These extensions become part of the application and remain in effect permanently. However, unlike Watchdogs [2], Exokernel does not allow an extension to be associated directly with a resource, e.g. to control file access.

The second method of extending Exokernel is to download code into the kernel. This model allows limited procedural extensibility. The safety of this code is assured through the use of a safe language or software fault isolation. These extensions persist until the application, on whose behalf the extension was installed, terminates.

## *Mach*

The Mach system was the pioneer of the microkernel architecture [1]. The system is comprised of a kernel that implements key abstractions such as tasks, threads, ports, messages, and memory objects, and a collection of user-level servers that implement high-level services, such as the file system. The system can be extended by replacing or modifying these user-level servers. This canonical microkernel is considered a user-level, extensible system, using hardware protection. Since extensions are implemented as user-level servers, Mach is a kernel-lifetime system with module-extensible granularity.

## *MS-DOS*

In MS-DOS, extensions are installed either as installable device drivers or as TSR (terminate and stay resident) programs. User programs, the kernel, and extensions share the same address space with no protection. At the time of initialization, an extension hooks itself into one or more of the system vectors to intercept system calls, calls to the ROM BIOS where some device drivers reside, and/or hardware interrupts. Extensions under MS-DOS can be very powerful since they are allowed to modify everything in the system. This open and uncontrolled extension interface gives MS-DOS tremendous flexibility at great cost; MS-DOS is unstable, sometimes difficult to configure, and offers no protection against hostile extensions such as computer viruses. The MS-DOS extension model is that of kernel-level, module extensibility with no protection (good will). Extensions persist until the kernel crashes (kernel lifetime) at which point they must be reinstalled.

## *μChoices*

μChoices is a micro-kernel reimplementation of the Choices operating system. The core of the system is a “nano-kernel,” which abstracts the underlying hardware of the physical machine, providing an idealized interface to the higher layers of the system. Traditional services are implemented using object-oriented frameworks, which decouple module interfaces and implementation[20]. μChoices also allows extensibility by means of downloadable software agents. These agents are implemented in a scripting language similar to Tcl, and are frequently used to aggregate multiple kernel calls, avoiding the cost of multiple crossings into and out of the operating system [4].

The use of an interpreted language to support extensibility places μChoices in the software-protected, kernel-level, kernel-lifetime extensible category. Since extensions are installed at the equivalent of system call interfaces, the granularity model is module-extensible.

## *Scout*

Scout is a parameterizable operating system targeted to communications-oriented applications [15]. A given Scout system is configured to provide just the functionality required for its target application domain—no more and no less. This technique has the advantage of providing a lean and efficient operating system for computer systems that perform specialized tasks, such as dedicated storage servers or web clients. The disadvantage of this approach is the lack of run time flexibility and customizability, limiting Scout’s utility as a general purpose operating system.

Scout is organized around the concept of a “path,” an operating system abstraction that encapsulates the flow of data from one part of the system to another, often crossing traditional operating system interface boundaries. System resources are allocated and scheduled on a per-path basis, and paths serve as the focal point for system optimizing.

Scout’s parameterizable architecture simplifies many of the design decisions associated with other extension technologies. Since all kernel code is fixed and compiled into the same address space, Scout falls in the procedural granularity, kernel-level, and “good will” protection categories. Extensions are permanent in that they will continue to execute until the kernel is reconfigured and rebuilt.

## *Solaris*

Solaris is an example of a conventional operating system; it is the UNIX operating system sold by SunSoft. It runs on the Sparc and Intel x86 architecture families. Solaris provides three different forms of extensibility: build-time kernel configuration, run-time downloading, and run-time modification of scheduling behavior.

Solaris provides build-time extensibility in the area of file systems (through the VFS layer) and scheduling (through scheduling classes). New file systems or scheduling classes that conform to the Solaris framework can be added to the kernel, even in the absence of kernel source code. At run-time, Solaris provides the super-user the ability to dynamically download device drivers (as

do most UNIX systems). Also at runtime, Solaris provides the super-user with the ability to dynamically assign processes to different scheduling classes (e.g. real-time class, time-sharing class) [21].

In all cases of Solaris extensibility, the extended functionality is provided at kernel-level, using no additional protection mechanisms (i.e. good will), and at the module granularity. The build-time additions fall into the category of parameterizable systems (since the code base and behavior are set at build time) and all extensions are permanent (i.e. they persist until the kernel is rebuilt). The downloading ability is an example of a kernel-lifetime, extensible feature, and the scheduling class selection is an example of an application-lifetime, reconfigurable feature.

### ***SPIN***

SPIN is an extensible operating system that allows applications to safely modify both its interface and implementation [3]. Extensions are written in a typesafe language (Modula-3) and downloaded into the kernel where they initiate a thread. Once installed, the thread can install handlers for any kernel events for which it has appropriate permission and in which it is interested. The use of Modula-3 to support a downloaded system places SPIN in the kernel-level, extensible, software-protected class. Extensions have kernel-lifetimes and are limited procedural extensible, in that they are restricted to those interfaces explicitly exported by the domains that implement particular services.

### ***VINO***

The goal of the VINO project [18] is to provide resource-intensive applications greater control over resource management. VINO supports the downloading of kernel extensions, which are written in C++ and protected using software fault isolation [22]. To facilitate graceful recovery from an extension failure, VINO runs each extension in the context of a transaction. If the extension fails or must be aborted (because it is monopolizing resources), the transaction mechanism undoes all actions taken by the extension. VINO extensions are bound with resources, including both applications and files, so extensions persist across system reboots and kernel upgrades. The VINO kernel is constructed from a collection of objects; extensibility is provided by replacing or overloading specific methods on those objects. The software-protected downloading of code places VINO in the kernel-level, software-protected, extensible category. Extensions have resource-lifetimes and are limited procedural extensible.

## **6. Conclusion**

The investigation of extensible operating systems is not a fundamentally new area. In practice, vendors have been selling extensible systems for a number of years. However, these systems have been plagued with a number of problems, most importantly, adaptability and reliability. The current trend we see in extensibility research is a synergy between the commercial and research sectors: the research community is addressing a number of problems with real systems. The variety of solutions to these problems are likely to provide tremendous insight into the future of operating systems in both arenas.

## 7. Bibliography

- [1] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M., “Mach: A New Kernel Foundation for UNIX Development,” *Proceedings of the Summer 1986 USENIX Conference*, July 1986, 93-112.
- [2] Bershad, B. N., Pinkerton, C. B. “Watchdogs—Extending the UNIX File System,” *Computing Systems*, 1, 2, Spring 1988, 169-188.
- [3] Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., Chambers, C., “Extensibility, Safety, and Performance in the SPIN Operating System,” *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995, 267–284.
- [4] Campbell, R., Tan, S., “μChoices: An Object-Oriented Multimedia Operating System,” *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, May 1995, 90-94.
- [5] Chen, B., Endo, Y., Chan, K., Mazieres, D., Dias, A., Seltzer, M., Smith, M., “The Measured Performance of Personal Computer Operating Systems,” *ACM Transactions on Computer Systems*, February 1996, 3–40.
- [6] Cowan, C., Walpole, J., Black, A., Inouye, J., Pu, C., Cen, S., “Adaptable Operating Systems,” *reference to paper in this same collection*.
- [7] Engler, D., Kaashoek, F., and O’Toole, J., “Exokernel: An Operating System Architecture for Application-Level Resource Management,” *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, December 1995, 251–266.
- [8] Fall, K., Pasquale, J., “Exploiting In-Kernel Data Paths to Improve Throughput and CPU Availability,” *Proceedings of the 1993 Winter Usenix Technical Conference*, San Diego, CA 1993, 327-334.
- [9] Fiuczynski, M., Bershad, B., “An Extensible Protocol Architecture for Application-Specific Networking,” *Proceedings of the 1996 USENIX Technical Conference*, San Diego, CA, 1996, 55–64.
- [10] Itoh, J., Yokote, Y., Tokoro, M., “SCONE: Using Concurrent Objects for Low-level Operating System Programming,” *Proceedings of the ACM OOPSLA’95*, October 1995.
- [11] Kiczales, G., des Rivières, J., Bobrow, D., *The Art of the Metaobject Protocol*, MIT Press, 1991.
- [12] Kiczales, G., Lamping, J., “Operating Systems: Why Object-Oriented?” *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*, Asheville, NC, December 1993, 25–30.

- [13] Microsoft Corporation, *MS-DOS SMARTDRV help page*, Redmond, WA, 1994.
- [14] Mogul, J., Rashid, R., Accetta, M., “The Packet Filter: An Efficient Mechanism for User-level Network Code,” *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, Austin, TX, November 1987, 39-51.
- [15] Montz, A., Mosberger, D., O’Malley, S., Peterson, L., Proebsting, T., Hartman, J., “Scout: A communications-oriented operating system,” Department of Computer Science, University of Arizona, Technical Report 94-20, June 1994.
- [16] Ousterhout, J., Cherenon, A., Douglis, F., Nelson, F., Welch, B., “The Sprite Network Operating System,” *IEEE Computer*, Vol. 21, No. 2, Feb. 1988, 23-36.
- [17] Ritchie, D., Thompson, K., “The UNIX Time-Sharing System,” *Communications of the ACM*, 17 7, July 1974, 365-375.
- [18] Seltzer, M., Endo, Y., Small, C., Smith, C. “An Introduction to the Architecture of the VINO Kernel,” Harvard University Center for Research in Computing Technology Technical Report 34-94, 1994.
- [19] Small, C., Seltzer, M. “A Comparison of OS Extension Technologies,” *Proceedings of the 1996 USENIX Technical Conference*, San Diego, CA, January 1996, 41–54.
- [20] Tan, S., Raila, D., Campbell, R., “An Object-Oriented Nano-Kernel for Operating System Hardware Support,” *Proceedings of the Fourth International Workshop on Object Orientations in Operating Systems*, Lund, Sweden, August 1995, 220-223.
- [21] Vahalia, U., *UNIX Internals, The New Frontiers*, Prentice-Hall, Inc., Upper Saddle River, NJ, 1996, 122–138.
- [22] Wahbe, R., Lucco, S., Anderson, T., Graham, S., “Efficient Software-Based Fault Isolation,” *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, December 1993, Asheville, NC, 175-188.