MSR-TR-2006-51
This is a draft paper that is under submission. Please contact Galen Hunt for citation information.

Sealing OS Processes to Improve Dependability and Security

Galen Hunt, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson,
James Larus, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, Brian Zill
Microsoft Research

Abstract

On most modern operating systems, a process is a hardware-protected abstraction for executing potentially mutable code and data. Common features of processes include: dynamic code loading, dynamic code generation, access to cross-process shared memory, and a universal API.

This paper argues that many of the dependability and security weaknesses of modern systems are exacerbated by this *open process architecture*. Moreover, this architecture impairs the ability of tools to analyze code statically, to improve its performance or dependability. By contrast, a *sealed process architecture* prohibits dynamic code loading, prohibits self-modifying code, prohibits shared memory, and replaces a universal API with a process-limited API. This paper describes an implementation of a sealed process architecture in the Singularity operating system, discusses its merits, and evaluates its impact. Among the benefits are: improved static program analysis, strong security guarantees, elimination of OS redundancies found in language runtimes such as the JVM and CLR, and better software engineering.

1. Introduction

The process, as a recognized operating system abstraction, debuted in Multics [52] in the 1960s. Multics processes included support for dynamic code loading, dynamic code generation, access to cross-process shared memory, and a universal API capable of directly modifying data in another process.

This *open process architecture* is now nearly universal. Aspects of this architecture, such as dynamic code loading and cross-process shared memory, were not in the early versions of UNIX [38] or early PC operating systems. However, their absences reflected limited address spaces and a lack of dynamic linking technology, and this “weakness” was quickly remedied. Today, for example, all four aspects of open processes are found in FreeBSD, Linux, Solaris, and Windows.

In the beginning, open processes gained popularity because they improved memory utilization by sharing library code across processes. Today, open process architecture’s most characteristic feature is the ease with which the OS or an application can be extended with plug-ins. New features and functionality are routinely loaded directly into these systems’ kernels and processes. For example, Microsoft Windows supports over 100,000 in-kernel device drivers, which enable it to control almost any hardware device. Other examples of widely used in-process extensions include dynamic content extensions in web servers (like ISAPI extensions to Microsoft’s IIS), extended stored procedures in databases, and web browser plug-ins.

1.1. Disadvantages of Open Processes

Although common, open processes are not “free.” This architecture has negative consequences for dependability, correctness, security, and performance.

Almost all open process systems use hardware protection [41] to ensure process isolation. All of the systems named above, and many others, rely on paged memory management and differentiated user and kernel instructions to ensure process isolation. Hardware-based process isolation is so common that it is generally assumed to be a “free” processor feature. Unfortunately, hardware-based isolation is not free. In Aiken *et al.* [3], we demonstrated performance costs from 2.5% (in a compute-bound task with no paging) to 33% (in an IPC bound task). Page table management and cache and TLB misses are responsible for this increase in execution time.

Moreover, the extension mechanism commonly associated with open processes is a major cause of software reliability, security, and backward compatibility problems. Although extension code is rarely trusted, verified, or even fully correct, it is loaded directly into a host’s process with no hard interface or boundary between host and extension. The outcome is often unpleasant. For example, Swift reports that faulty device drivers cause 85% of diagnosed Windows system crashes [47]. Moreover, because an extension lacks a hard interface, it can use private aspects of its host’s implementation, which can constrain evolution of a program and require extensive testing to avoid future incompatibilities.

Dynamic code loading imposes a less obvious tax on performance and correctness. Software that can load code is an open environment, in which it is impossible to make sound assumptions about the system's states, invariants, or valid transitions. Consider the Java Virtual Machine (JVM). An interrupt, exception, or thread switch can invoke code that loads a new file, overwrites class and method bodies, and modifies global state [45]. In general, the only feasible way to analyze a program running under such conditions is to start with the unsound assumption that the environment will not change arbitrarily between any two operations.

Open processes complicate security. If the code running in a process is not known *a priori*, it is difficult to rationalize subsequent access control decisions based on code identity. Perhaps for this reason it is rare for operating systems to provide any strong guarantees about application identity, relying instead on the identity of an authenticated user or an application masquerading as a user. Software attestation solutions like Terra [20] and NGSCB [35] must use hardware and virtual machines to prevent modifications to the code of running processes.

1.2. Contributions

This paper describes a new *sealed process architecture* in which the OS guarantees that the code in a process cannot be altered once the process starts executing. The architecture prohibits dynamic code loading, prohibits self-modifying code, prohibits cross-process sharing of memory, and replaces a universal API with a process-limited API. Of particular import, the sealed architecture incorporates a single extension mechanism for both OS and applications: extension code executes in a new process, distinct from its host.

Sealed processes offer many advantages. They increase the potential of static program analysis to improve performance and reliability. They enable stronger security guarantees. They allow the elimination of duplication of OS-level features in virtual execution environments such as the Sun's JVM and Microsoft's Common Language Runtime (CLR). Finally, they encourage better software engineering.

We have implemented sealed process in the Singularity operating system. Singularity is a new system being developed as a basis for more dependable system and application software [28, 29]. Singularity exploits advances in programming languages and tools to create an environment in which software is more likely to be built correctly, program behavior is easier to verify, and run-time failures can be contained.

The rest of the paper is organized as follows. Section 2 describes the sealed process architecture and discusses its qualitative merits. Section 3 describes the implementation of sealed processes in Singularity. Section 4 contains a quantitative evaluation of the sealed process architecture in Singularity. Section 5 describes related work and Section 6 contains conclusions and a discussion of future work.

2. Sealed Process Architecture

A *sealed process architecture* is an architecture in which the operating system guarantees that the code of a process cannot be altered once the process starts executing and in which the state of a process cannot be directly modified by another process.

A *sealed kernel* is an OS kernel whose code cannot be altered after system boot and whose state cannot be modified by any process. An operating system with an open kernel could provide sealed processes. However, in this paper, we shall assume that a sealed process architecture also implies a sealed kernel.

We can draw an instructive analogy between sealed processes and sealed classes in object-oriented languages. Practice has shown that designers and implementers of classes need a mechanism to limit class extension and force extensions to use a declared interface [8] in order to avoiding errors that typically appear only at runtime. Similarly, by running dynamic extensions in a separate process, the sealed process architecture increases system dependability by forcing application and system extensions to use declared interfaces.

2.1. Sealed Process Invariants

The sealed process architecture maintains four invariants:

- **The fixed code invariant:** Code within a process cannot be altered once the process starts execution.
- **The state isolation invariant:** Data within a process cannot be directly accessed by another process.
- **The explicit communication invariant**: All communication between processes must occur through explicit mechanisms, with explicit identification of the sender, and explicit receiver admission control over incoming communication.
- **The closed API invariant**: The API between a process and the system must maintain the fixed

code, state isolation, and explicit communication invariants.

The fixed code invariant ensures that sealed processes are closed code spaces. Code in a process may come from one executable or from a collection of executables and libraries, but it is fully linked and loaded before execution starts. As a consequence, a process cannot dynamically load or generate code into its own process. Because code is known *a priori*, and can be checked against certificates provided by the code's publisher, it is possible to reason about the access rights of a process using code identity. For example, the security principal inherent in a process instance might include an authenticated user, a program, and/or a publisher.

The state isolation invariant ensures that code in a process reads and modifies data only in itself. This invariant disallows communication through shared memory. Processes can communicate through memory, but ownership and access to data must be handed off between the processes so that one process is not able to change the contents of the memory while another is reading it.

The explicit communication invariant ensures that all communication is subject to admission control policies. Implicit or anonymous communication channels, such as shared memory or manipulation of a Windows message pump, are forbidden. Furthermore, the explicit communication invariant ensures that both the kernel and recipient process have the ability to accept or deny requests to establish a communication channel. These constraints allow a process to create a communication channel to itself and pass that channel (and its implicit communication right) to another process. Similarly, a process can receive a channel from one process and hand it off to a third process. Finally, the explicit communication invariant allows a process to invest one or more communication rights into a child process at creation time.

In general, the explicit communication invariant ensures that a process can only communicate with the transitive closure of processes reachable through its existing communication graph (or extensions of the graph caused by creation of child processes). In practice, intermediate processes (including the kernel) and the OS communication mechanisms can further restrict the communication graph. For example, an intermediate process can refuse to forward a communication request based on access control.

The closed API invariant ensures that the OS API provided to an unprivileged process does not include a mechanism that could subvert the fixed code, state isolation, or explicit communication invariants. For example, the closed API invariant ensures that the base API does not include a mechanism to write to the memory of any other process.

Most open process systems include debugging APIs that allow reading and writing of another process's memory. Strictly speaking, the closed API invariant does not prohibit debugging APIs, but it does prohibit providing debugging APIs to unprivileged processes. Preferably debugging capabilities should be available in conjunction with specific communication rights and only after both the kernel and the target process (or the certifier of the target process) have agreed that debugging access is appropriate.

2.2. Qualitative Benefits

The sealed process architecture increases the accuracy and precision of static program analysis. The fixed code invariant allows a static analysis tool to safely assume that it has knowledge of all code that will ever execute in a process. For example, in a sealed process, a whole-process optimizing compiler can perform aggressive inter-procedural optimization, such as eliminating methods, fields, and even whole classes that are unreachable within a specific program.

This aggressive optimization cannot be performed *safely* in open process architectures. Imagine, for example, aggressively optimizing an open OS kernel that support dynamically loaded device drivers. A compiler might remove an apparently unused data field, thus reducing the size of a data structure, only to later have an unfortunate user load a driver that accesses this field.

The soundness of program correctness tools, such as Microsoft's Static Driver Verifier [5], is limited by open process architectures. Systems of this sort make an essential assumption that they can soundly approximate a program's worst case behavior since they analyze all of its code. If a tool cannot analyze all code, its approximations may be incorrect. Open systems are verifiable only if their extension points are fully specified so that an analysis tool can conservatively approximate the behavior of missing code. The extension points of open processes, because they expose the entire internal code of a process, are underspecified and therefore inherently unsound.

Sealed processes enable stronger security guarantees than open architectures. Code-based security systems such as Microsoft Authenticode and Java [23] attempt to make security guarantees by validating the signature of a program residing *on disk*. However, none of these guarantees hold once the code is loaded into an open process and mutated. With a sealed process, the certification of disk contents can be extended to the executable contents of a process. A sealed architecture can guarantee that a program will execute in its certified form because of the state isolation and closed API invariants. No open process architecture can make such a claim. When coupled with hardware protection for external tampering, such as NGSCB [35], sealed processes enable an execution model in which a process is a trustable entity.

Sealed processes enable the elimination of the OS redundancies found in virtual execution environments such as the JVM and CLR. A sealed process system needs only one error recovery model, one communication mechanism, one security policy, and one programming model because the sealed process architecture uses a single mechanism for both protection and extensibility instead of the conventional dual mechanisms of processes and dynamic code loading. As a consequence, the confusing layers of partially redundant mechanisms and policies found the CLR and JVM can be removed. For example, our experiments show that the compiled CLR base class library would be 30% smaller without the checks and metadata required for its security model meant to compensate for the weaknesses of open processes.

Sealed processes encourage—but obviously cannot guarantee—that programmers practice better software engineering by encouraging modularity and abstraction. In a sealed architecture, OS and application extensions, such as device drivers and plug-ins, can communicate only through well-defined interfaces. The process boundary between host and extension ensures that an extension interface cannot be subverted.

At least for the kernel, the benefits of a sealed architecture have long been recognized by the OS community. For example, microkernels, such as Mach [2], L4 [24], SPIN [7], VINO [42], and the Exokernel [12], recognize that an OS kernel would be more reliable if it was extended in separate processes. These microkernels realized benefits from closing the kernel, but none generalized the principle into a sealed process architecture.

2.3. Limitations

Although a sealed architecture offers benefits, it also imposes limitations. The set of programs that can execute on a sealed architecture is limited by design. For example, the fixed code invariant prohibits self-modifying code and the state isolation invariant prohibits communication through shared memory.

2.3.1 Programmability

Communication via memory shared between processes is notoriously prone to concurrency bugs. A sealed process OS may prevent this class of errors, but the message passing is also a common source of programming errors. These errors include marshalling code that breaks type-safety properties and communication protocol violations that lead to deadlocks and livelocks. These problems can be mitigated by programming language extensions that concisely specify communication over channels and by verification tools [15].

Sealed processes also increase the complexity of writing program extensions, as the host program's developer must define a proper interface that does not rely on shared data structures. The extension's developer must program to this interface and possibly re-implement some functionality from the parent. Nevertheless, the widespread problems inherent in dynamically loaded extensions argue for alternatives that increase isolation between an extension and its parent. Singularity's demonstrates that an out-of-process extension mechanism works for applications as well as system code; does not depend on the semantics of an API, unlike domain-specific approaches such as Nooks [47]; and provides simple semantic guarantees that can be understood by programmers and used by tools.

2.3.2 Dynamic Code Generation

The inability to generate code dynamically into a running sealed process precludes common coding patterns such as just-in-time compilation and the online translation of high-level abstractions such as regular expression. In a sealed architecture, such usage must be implemented by generating the code into a separate process image, starting this process, and communicating with the process over a well-defined communication channel. Note that interpreters for any language can still be written in the sealed process architectures.

2.3.3 Data Sharing

In an open process OS, shared libraries and DLLs are a common way to reduce the code footprint of the system. For example, a typical Windows Sever 2003 system running a mix of user applications shows 74 processes using a total of 5.4GB of virtual address space, but only 760MB of private data/code. In an open process, the extensions can be co-located into a single process, thus sharing both read-only and read-write data. In a sealed architecture, multiple extensions may share read-only data and code from a common library, but they can’t share read-write data.

2.3.4 Scheduling Overhead

Since a sealed process can only be extended by creating a child process that provides the additional functionality via an IPC interface, it is expected that there will be more processes involved in servicing any given request. These additional processes not only introduce communication overheads such as data copying, but exacerbate the scheduling problem faced by the kernel. The prevalence of server processes multiplexed between multiple clients could make resource accounting more difficult, and the increased number of (non-blocked) threads may preclude sophisticated scheduling algorithms.

In practice, we have found that a typical Singularity system contains a large number of threads (hundreds of threads are not uncommon), but the number of non-blocked threads at any time is usually quite small. However, this may not be true for all sealed architectures.

3. Singularity

Singularity is a new system under development as a basis for more dependable system and application software. Singularity incorporates two important technologies to significantly reduce the cost of a sealed process architecture: *software isolated processes* (SIPs) and first-class language support for inter-process communication.

SIPs use programming language safety and the invariants of the sealed process architecture to provide a less expensive process isolation mechanism than hardware protection. All untrusted code running in a SIP must be written in a safe language and statically verified to be type safe. Due to verified memory safety, SIPs can share the same hardware protection domain and in fact execute in privileged mode with the kernel.

SIPs are inexpensive to create. Low cost makes it practical to use SIPs as a fine-grain isolation and extension mechanism. Versions of Singularity with and without hardware memory protection allow direct comparison of performance overheads of isolation mechanisms. Without hardware isolation, system calls and inter-process communication run significantly faster (30–500%) and communication-intensive programs run up to 33% faster [3].

SIPs communicate through a bidirectional type-safe message-passing mechanism called a *channel*. Each channel has exactly two endpoints. At any point in time, each channel endpoint is owned by at most one thread. High-level message primitives are provided to transfer typed data structures from the object space of one process to another without violating type safety or compromising the integrity of the respective garbage collectors and language runtimes. In fact, bulk data transfer can be achieved without any copying because static verification (made possible by the sealed process abstraction) prevents a sending thread from accessing objects it no longer owns. Fähndrich *et al.* [13] contains a full description of the first class language support and type system for IPC in Singularity.

The contributions of SIPs and first IPC support make the sealed process architecture viable. In the remainder of this section we describe how the architecture works

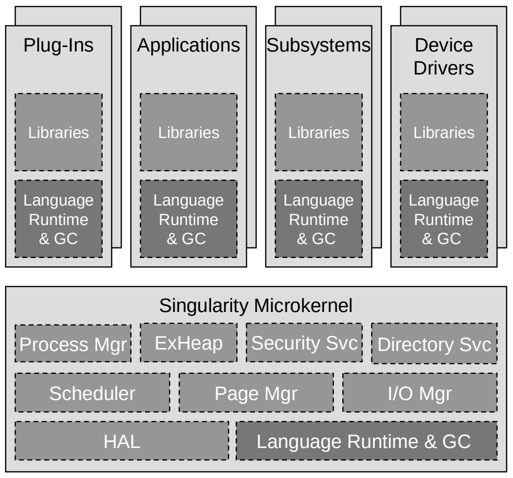

Figure 1. Singularity System Architecture.

3.1. Singularity Components

Figure 1 depicts the key components of the Singularity OS. The microkernel provides the core functionality of the system, including paged memory management, process creation and termination, communication through channels, scheduling, I/O, security, and a local directory services. Most of the Singularity’s functionality and extensibility exists in processes outside of the kernel. In particular, all subsystems and devices drivers run in separate processes.

Unlike previous systems that relied on language safety, Singularity SIPs execute independently. Each SIP contains its own memory pages, language runtime, and garbage collector. Due to the state isolation invariant, the language runtime and garbage collector can employ data layout and GC algorithms most appropriate for a particular process. SIPs are created and terminated by the operating system, so that on termination, a SIP's resources can be efficiently reclaimed.

The Singularity kernel and language runtimes consist almost entirely of safe code. The rest of the system consists of only verifiably safe code, including all device drivers, subsystems, applications, and plug-ins. While all untrusted code *must* be verifiably safe, parts of the Singularity kernel and language runtime, called the *trusted base*, are not verifiably safe. Language safety protects this trusted base from untrusted code.

In addition to the message-passing mechanism of channels, processes communicate with the kernel through a limited API that invokes static methods in kernel code. This interface isolates the kernel and process object spaces. All parameters to this API are values, not pointers, so the kernel and process's garbage collectors need not coordinate. The Singularity API maintains the closed API invariant. Only two API calls affect the state of another process. The call to create a child process specifies the child's code manifest and gives an initial set of channel endpoints *before* the child process begins execution. The call to stop a child process destroys its state *after* all threads have ceased execution.

3.1.1 Trusted Base

Code in Singularity is either *verified* or *trusted*. Verified code's type and memory safety is checked by at install time. Unverifiable code must be trusted by the system and is limited to the HAL, kernel, and parts of the run-time system. Most of the kernel is verifiably safe, but small portions are written in assembler, C++, and unsafe C#.

All code outside the trusted base is written in a safe language, translated to safe Microsoft Intermediate Language (MSIL)[1], and then compiled to x86 by the Bartok compiler [16] at install time. Currently, we trust that Bartok correctly verifies and generates safe code. This is obviously unsatisfactory in the long run and we are working on using typed assembly language to verify the output of the compiler and reduce this part of the trusted computing base to a small verifier [32]

[1]MSIL is the CPU-independent instruction set accepted by the Microsoft CLR. Singularity uses the MSIL format. Features specific to Singularity are expressed through metadata extensions in the MSIL.

3.1.2 Scheduler

The Singularity scheduler is optimized for a large number of threads that communicate frequently. The scheduler maintains two lists of runable threads. The first, called the *unblocked* list, contains threads that have recently become runable. The second, called the *preempted* list, contains runable threads that have been pre-empted. When choosing the next thread to run, the scheduler removes threads from the unblocked list in FIFO order. When the unblocked list is empty, the scheduler removes the next thread from the preempted list. Whenever a scheduling timer interrupt occurs, all threads in the unblocked list are moved to the end of the preempted list, followed by the thread that was running when the timer fired. The first thread from the preempted list is scheduled and the scheduling timer is reset.

The net effect of the two list scheduling policy is to favor threads that are awoken by a message, do a small amount of work, send one or more messages to other processes, and then block waiting for a message. This is a common behavior for threads running message handling loops.

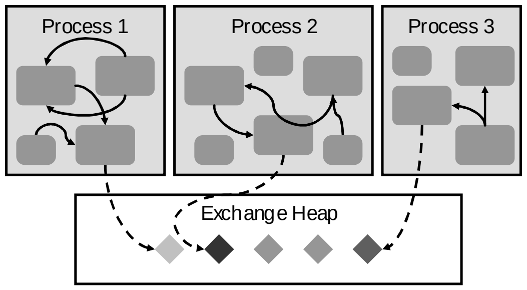

Figure 2. The Exchange Heap.

3.1.3 Exchange Heap

The Exchange Heap, which underlies efficient communication in Singularity, holds data passed between processes. The Exchange Heap is not garbage collected, but instead uses reference counts to track usage of blocks of memory (Figure 2). Allocations within the Exchange Heap are owned by at most one process at time with ownership enforced by static verification. Allocations may be split; for example, protocol processing code in a network stack can strip protocol headers off a packet and hand the payload to an application without copying the packet.

3.1.4 Processes

In the most common deployment, a Singularity system lives in a single address space without virtual memory addressing.[2] The common address space is logically partitioned into pages for a kernel object space, pages for an object space for each process, and pages for the Exchange Heap. The kernel does not have pointers into process object spaces, nor does any process have a pointer to another process' objects. Adherence to the state isolation invariance ensures that each process can be garbage collected and terminated independently.

A process starts with a single thread, enough memory to hold its code, an initial set of channel endpoints, and a small heap. It obtains additional memory by calling the kernel's page manager, which returns new, unshared pages. These pages need not be adjacent to the process's existing address space, since the garbage collectors do not require contiguous memory regions.

A process can call kernel APIs functions to create and start additional threads. Singularity uses linked stacks to reduce the memory overhead of a thread. These stacks grow on demand by adding non-contiguous segments of 4KB or more. Bartok performs static interprocedural analysis to optimize placement of overflow tests [51].

To reduce the overhead of API calls, Singularity does not switch stacks when a process calls the kernel. Instead, the Singularity runtimes uses stack markers to track the ownership of stack frames so that the kernel's GC can traverse kernel frames and the process's GC can traverse process frames. These markers also facilitate terminating processes cleanly. When a process is killed, a kernel exception is thrown in each of its threads, which skips over and frees the process's stack frames.

Processes are created from a process manifest [43]. The manifest describes the process in terms of its code, its resources and its dependencies on the kernel and on other processes. All code within a Singularity process must be listed in the process manifest. Although many existing application setup descriptions combine declarative and imperative aspects, Singularity manifests are unique in that they contain only declarative statements that describe the desired state of the application after installation or update.

[2] Singularity supports multiple address spaces and hardware protection through user-configurable runtime mechanism called a *protection domain*. However, software isolation is preferred in most cases as it is secure and avoids the penalties (2.5% to 33%) of hardware protection.

3.1.5 Channels

Singularity processes communicate exclusively by sending messages over channels. Channel communication is governed by statically verified channel contracts that describe messages, message argument types, and valid message interaction sequences as finite state machines. Messages are tagged collections of values or message blocks in the Exchange Heap that are transferred from a sending to a receiving process. These primitives enforce much stronger semantics than the low-level IPC mechanisms of a typical microkernel.

Channel endpoints can be sent in messages over channels. Thus, the communication network can evolve dynamically while conforming to the explicit communication invariant. Sending and receiving on a channel requires no memory allocation. Sends are non-blocking and non-failing; receives block synchronously until a message arrives or the send endpoint is closed.

A process creates a channel by invoking a contract's static `NewChannel` method, which returns the channel's two endpoints. The process can pass either or both endpoints to other processes over existing channels.

When data or endpoints are sent over a channel, ownership passes from the sending process, which may not retain a reference, to the receiving process. This ownership invariant maintains the state isolation invariant and is enforced by the language using linear types and by the run-time systems.

3.1.6 Garbage Collection

Garbage collection is an essential component of most safe languages, as it prevents memory deallocation errors that can subvert safety guarantees. In Singularity, kernel and process object spaces are garbage collected.

Experience and the large number of garbage collection algorithms strongly suggest that no one garbage collector is appropriate for all applications [17]. Singularity's sealed process architecture decouples the algorithm, data structures, and execution of each process's garbage collector. Each process can select a GC to accommodate its objectives and to run without global coordination. The three aspects of Singularity that make this possible are: each sealed process has its own runtime; pointers do not cross process or kernel boundaries, so collectors need not consider cross-space pointers; and messages on channels are not objects, so agreement on memory layout is only necessary for data in the Exchange Heap.

3.1.7 Security and Access Control

Processes receive an immutable security principal name at creation. A principal name is an ordered list of applications, each of which can act in specific roles. The list ordering is intended to reflect the chain of application invocations that led to the creation of the named process. Human users are represented as roles of programs trusted to perform user authentication. The design for our *compound principal* grammar has been previously discussed [1].

Because security principal names have structure, we cannot use simple integers or groups of names to specify which principals can access which resources. Instead, Singularity access control lists are regular expressions. Common subexpressions (the equivalent of groups in conventional access control systems) can be named in and expanded from the Directory Service. The structure of compound principals makes it appealing for intermediate nodes in a security-relevant operation to simply record their participation and let the ultimate reference monitor decide. Because of this, impersonation is largely unnecessary and we do not support it in Singularity.

Access control in Singularity is discretionary. Programs that control resources do so by means of explicit access control checks. Since all communication takes place over controlled channels, the subject of such access control checks can be determined by examining the message source principal associated with the incoming channel. Access control decisions are implemented by means of a security library that is bound into every SIP that guards resources. The Singularity kernel provides minimal support by way of a service that maps the shorthand principal identifiers used by the channel implementation into full names.

One resource of particular interest is the Directory Service. This service implements a naming tree whose names are used by applications to establish communications channels with system components such as files, services, and devices. The Directory Service, then, employs access checks to allow system policy to regulate which principals can register which names, and, for any given name, which principals can establish channels to it. Thus, while processes could in theory authenticate their channel partners for each new channel, in practice it is sufficient to assume that the Directory Service is trusted to enforce system channel establishment policy.

In the current Singularity implementation, any one process can speak for only one security principal, and this principal cannot be changed. This model is appealing in that it leaves little latitude for *confused deputy attacks* or other attacks that depend on security-relevant code dealing (incorrectly) with the authority of multiple different security principals. However, even with the low overhead on process invocation that Singularity provides, it may be over-optimistic to assume that all code dealing with multiple principals can be avoided. We expect to introduce controlled delegation of authority between processes in the form of specially designated communications channels. This is the subject of ongoing research.

4. Quantitative Evaluation

The sealed process architecture represents a significant change from the open processes generally in use. In this section we evaluate the merits of the sealed process architecture by qualitatively comparing the performance of the current version of Singularity primarily with Windows Server 2003 R2. It is important to note that the comparison is not balanced. Windows is a mature, feature rich system, while Singularity has been in development as a research prototype for just over two years. The evaluation in this section should be considered proof of viability more than a definitive analysis of the sealed process architecture.

All experiments were run on an AMD Athlon 64 3000+ (1.8 GHz) CPU with an NVIDIA nForce4 Ultra chipset, 1GB RAM, a Western Digital WD2500JD 250GB 7200RPM SATA disk (command queuing disabled), and the nForce4 Ultra native Gigabit NIC (TCP offload disabled). Versions of systems used were FreeBSD 5.3, Red Hat Fedora Core 4 (kernel version 2.6.11-1.1369_FC4), and Windows Server 2003 R2.

4.1. SPECweb99

To quantify the overhead of the sealed process architecture, we measured the performance of Singularity and Windows running the SPECweb99 benchmark [44]. As designed, the SPECweb99 benchmark measures the maximum number of simultaneous connections a web server can support, while maintaining a specified minimum bandwidth on each connection. The benchmark consists of both static and dynamic content. Static content is selected using a Zipf distribution consisting of 35% files smaller than 1KB, 50% files larger than 1KB and smaller than 10KB, 14% files larger than 10KB and smaller than 100KB, and 1% files larger than 100KB and smaller than 1MB.

The Singularity implementation uses six sealed process: a NIC driver, the TCP/IP stack, the HTTP server, the SPECweb99 content plug-in, the file system, and a disk driver (see Figure 3). As required for formal benchmarking, the Singularity implementation of SPECweb99 is not fully conformant. In particular, our HTTP server doesn't support logging and our TCP/IP stack does not fully support the IPv4 sliding window protocol. In addition, we use a smaller execution time on both Windows and Singularity than is required for formal SPECweb99 results.

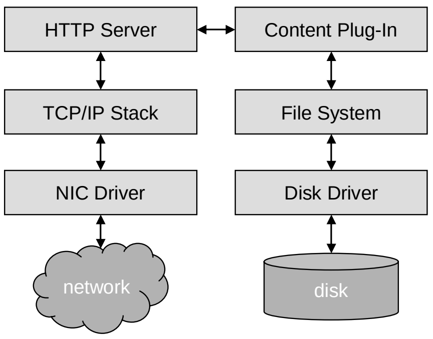

Figure 3. SPECweb99 running as six sealed processes.

The Windows Server 2003 implementation of SPECweb99 runs on IIS 6.0 and takes advantage of all optimizations available in an open process architecture (see Figure 4). In this implementation, the NIC driver, TCP/IP stack, disk driver, file system, and HTTP dispatcher code are all loaded into the Windows kernel using the device driver extension model. The HTTP dispatcher transfers content directly from the file system (or file system cache) to the TCP/IP stack without leaving the kernel. Dynamic content requests travel directly from the `http.sys` driver in the kernel to the IIS worker process, which contains the dynamic content plug-in running as an ISAPI extension. The `inetinfo.exe` controller process is executed only on the first dynamic content request to start the worker process.

Singularity achieves 247 ops/second with a weighted average throughput of 376 Kbits/second. By contrast, Microsoft Windows 2003 running the IIS web server, on identical hardware, achieves 761 ops/second with a weighted average throughput of 336 Kbits/second. Singularity's average response time, with 78 connections, of 320 ms/op is comparable to Window's time of 304 ms/op. Singularity's throughput on this benchmark is not bound by the sealed architecture, but is disk bound and limited by the caching algorithms in our experimental file system. In contrast, Windows is network bound thanks to its highly tuned file caching. We have an ongoing effort to improve the file system and expect much closer performance in the future. While definitely not conclusive, the response time numbers suggest that the sealed process architecture is viable.

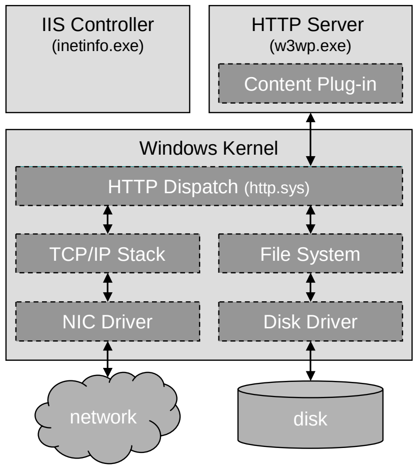

Figure 4. SPECweb99on Windows Server 2003

4.2. Improved Static Analysis

Sealed processes offer improved opportunities for static analysis because all of the code that will run in a process is known before the process begins execution. Static analysis is available to any process architecture, but sound static analysis of a complete process is possible only when the entire process code is fixed.

Program	Whole	w/ Tree Shake	% Reduction
Kernel	2371 KB	1291 KB	46%
Web Server	2731 KB	765 KB	72%
SPECweb99 Plug-in	2144 KB	502 KB	77%
Ide Disk Driver	1846 KB	455 KB	75%

Table 1. Reduction in code size via tree shaking enabled with seal processes.

One example of the type of static analysis enabled by sealed processes is whole process tree shaking. The Bartok compiler creates a tree of all of the code available within a process. It then *safely* eliminates (a.k.a. shakes out) fields, methods, and classes unused in all possible executions of the process. As shown in Table 1, tree shaking can reduce program code size by as much as 75%. Most importantly, tree shaking of extensible programs, such as the web server can reduce code size by as much as 72%. The latter is important because without sealed processes, the compiler could not remove code because it wouldn't know which code might be required by future plug-ins. Even though plug-ins must include their own libraries (as they run in separate processes), the combined code size of the web server and the SPECweb99 plug-in with tree shaking is still 54% smaller than just the web server without tree shaking.

Other static analysis tools and techniques benefit from sealed processes. For example, Bartok static checks that methods flagged with `[NoAlloc]` do not invoke any code that might perform a heap allocation. This check is useful for verifying that portions of the kernel, such as interrupt handlers do not allocate memory.

4.3. Program Complexity

The sealed architecture replaces access to shared memory and shared functions with explicit message passing. Hosts and extensions in sealed processes must incorporate code for communicating; in an open architecture they could directly access each other's state. Developers must now write contracts, explicit communication code, and some routines inaccessible through the host's published interface. On the other hand, communication is now sufficiently explicit to be statically verified.

Code Description	Lines	% of Orig.
Original web server	1486	100%
New host code	263	18%
New channel contract	52	3%
New extension code	76	5%
Total	**1877**	**126%**

Table 2. Added code for explicit communication.

Table 2 shows the additional cost of explicit communication in the Cassini web server. The original Cassini web server was implemented on the CLR and exchanged state with plug-ins through a shared property bag structure for each HTTP request. On Singularity, Cassini uses two contracts (one for page requests and one for the HTTP properties). We added 263 lines of IPC code in the web server, 52 lines of channel contract, and 76 lines of extension stub, for a total increase of 391 lines (26%) over the original.

While a 26% increase in code is non-trivial, our experience suggests that this may be an upper bound. In practice, the brunt of additional code is paid by the host developer and re-used across many extensions. For example, of 9445 lines of device driver code in Singularity, 1597 lines (17%) are related to interprocess communication with either client processes or the I/O subsystem.

	Cost (in CPU Cycles)			
	API Call	Thread Yield	Message Ping/Pong	Create Process
Singularity	80	365	1,041	388,162
FreeBSD	878	911	13,304	1,032,254
Linux	437	906	5,797	719,447
Windows	627	753	6,344	5,375,735

Table 3. Cost of basic operations.

4.4. Costs of Primitive Operations

Table 3 reports the cost of primitive operations in Singularity and three other systems. For each system, we conducted an exhaustive search to find the cheapest API call [28]. The FreeBSD and Linux "thread yield" tests use user-space scheduled pthreads as kernel scheduled threads performed significantly worse. Windows and Singularity both used kernel scheduled threads. The "message ping pong" test measured the cost of sending a 1-byte message from one process to another. On FreeBSD and Linux, we used sockets, on Windows, a named pipe, and on Singularity a channel with a single message argument.

Message Size (bytes)	CPU Cycles		
	Singularity	Linux	Windows
4	1316	5544	6641
16	1267	5379	6600
64	1282	5549	6999
256	1271	5519	7353
1024	1267	5971	10303
4096	1274	8032	17875
16384	1275	19167	47149
65536	1268	87941	187439

Table 4. IPC costs.

A basic thread operation, such as yielding the processor, is roughly three times faster on Singularity than the other systems. ABI calls and cross-process operations run significantly faster (5 to 10 times faster) than the mature systems because of Singularity's SIP architecture.

Singularity's process creation time is significantly lower than the other systems because SIPs don't need MMU page tables and because Singularity does not need to maintain extra data structures for dynamic code loading. Process creation time on Windows is significantly higher than other systems because of its extensive side-by-side compatibility support for dynamic load libraries.

Singularity use first class language support to achieve zero-copy communication between SIPs [14]. Soundness of zero-copy semantics are verified by static analysis on the entire contents of sealed process. Table 4 shows the cost of sending a payload message from one process to another on Singularity, Linux, and Windows for comparison.

5. Related Work

The large amount of related work can be divided to two major areas: extension isolation and OS architecture and.

5.1. Extension Isolation

Short of providing sealed processes, there have been many attempts to alleviate problems caused by open processes, through providing protection and isolation mechanisms within a process, most typically for extensions.

Device drivers are both the most common operating system extension and the largest source of defects [10, 33, 47]. Nooks provides a protected environment in the Linux kernel to execute existing device drivers [46, 47]. It uses memory management hardware to isolate a driver from kernel data structures and code. Calls across this protection boundary go through the Nooks runtime, which validates parameters and tracks memory usage. Singularity, without the pressure for backward compatibility, provides mechanisms (SIPs and channels) that are general programming constructs, suitable for application and system code, as well as for device drivers.

Software fault isolation (SFI) isolates code in its own domain by inserting run-time tests to validate memory references and indirect control transfers, a technique called sandboxing [53]. Sandboxing can incur high costs and only provides memory isolation between a host and an extension. It does not offer the full benefits of language safety for either the host or extension. Sandboxing also does not control data shared between the two, so they remain coupled in case of failure.

Sun's JVM and Microsoft's Common Language Runtime (CLR) are virtual execution environments that attempt to compensate for some weaknesses of open processes by using fine-grain isolation and security mechanisms. Both are open environments that encourage dynamic code loading (e.g., Applets) and run-time code generation. They use language safety as their protection mechanism, but must introduce complex security mechanisms and policies, such as Java's fine grain access control or the CLR's code access security, to prevent code from accessing system internals and expressive interfaces [34]. These mechanisms are difficult to use properly and impose considerable overhead. Singularity runs extensions in separate sealed processes, which provide a stronger assurance of isolation and a more tractable security problem that does not entail a large number of fine grain policy decisions.

Computations sharing an execution environment are not isolated upon failure. A shared object can be left in an inconsistent or locked state when a thread fails [18]. When a program running in a JVM fails, the entire JVM process typically is restarted because it is difficult to isolate and discard corrupted data and find a clean point to restart the failed computation [9].

Other projects have implemented OS-like functionality to control resource allocation and sharing and facilitate cleanup after failure in open environments. J-Kernel implemented protection domains in a JVM process, provided revocable capabilities to control object sharing, and developed clean semantics for domain termination [25]. Luna refined the J-Kernel's run-time mechanisms with an extension to the Java type system that distinguishes shared data and permits control of sharing [26]. KaffeOS provides a process abstraction in a JVM along with mechanisms to control resource utilization in a group of processes [4]. Java has incorporated many of these ideas into a new feature called isolates [36] and Microsoft's CLR has had a similar concept called AppDomains since its inception.

Singularity eliminates the duplication between an operating system and these run-time systems by providing a consistent mechanism across all levels of the system. Singularity's SIPs are sealed and non-extensible, which provides a greater degree of isolation and fault tolerance than Java or CLR approaches.

5.2. OS architecture

Singularity is a microkernel operating system that differs in many respects from previous microkernel systems, such as Mach [2], L4 [24], SPIN [7], VINO [42], Taos/Topaz [50], and the Exokernel [12]. Microkernel operating systems partition the components of a monolithic operating system kernel into separate processes, to increase the system's failure isolation and reduce development complexity. Singularity generalizes this sound engineering methodology (modularity) to the entire system, by providing lightweight processes and inexpensive interprocess communication, which enable an application to be partitioned and still communicate effectively.

Previous systems did not seal the kernel or processes. Hardware-enforced process isolation has considerable overhead, and so microkernels evolved to support kernel extensions, while developing mechanisms to protect system integrity. SPIN was closest to our approach, as its extensions were written in a safe language and relied on language features to restrict access to kernel interfaces [7]. Vino used sandboxing to prevent unsafe extensions from accessing kernel code and data and lightweight transactions to control resource usage [42]. However, both systems allowed extensions to directly manipulate kernel data, which left open the possibility of corruption through incorrect or malicious operations and inconsistent data after extension failure. Exokernel defined kernel extensions for packet filtering in a domain-specific language and generated code in the kernel for this safe, analyzable language [19].

Other operating systems have been written in safe programming languages. Early “open” systems [30] were diametrically opposed to Singularity, as they encouraged dynamic code loading. These systems were developed as “single user” systems, and consequently paid little attention to security, isolation, or fault tolerance. Smalltalk-80 and Lisp Machine Lisp used dynamic typing and run-time checking to ensure language safety, but isolation depended on programmer discipline and could be subverted through introspective and system operations [21, 54]. Pilot and Cedar/Mesa were single-user, single-address space systems implemented in Mesa, a statically typed, safe language [37, 48]. More recently, Inferno is a single address space operating system that runs programs written in a safe programming language (Limbo) [11].

More recent safe systems provided processes. RMoX is an operating system partially written in occam [6]. Its architecture is similar to Singularity, with a system structured around message-passing between processes. However, RMoX uses a kernel written in C and only its device drivers and system process are written in a safe language.

Taos/Topaz [49] was a multiple address space, microkernel operating system written in Modula2+ [39]. Taos could emulate a Unix interface for programs written in unsafe languages [31], but a rich set of APIs as well as a lightweight RPC mechanism were available for the type-safe environment. Shared-memory artifacts were apparent in several aspects of the Taos implementation. In contrast to synchronous RPC, Singularity uses asynchronous message passing over strongly typed channels, which is more general (RPC is a special case) and permits verification of communication behavior and system-wide liveness properties.

Several operating systems have been written in Java. JavaOS is a port of the Java virtual machine to bare hardware [40]. It replaces a host operating system with a microkernel written in an unsafe language and Java code libraries. JavaOS supports a single open process shared between all applications.

The JX system is similar to Singularity in many respects. It is a microkernel system written almost entirely in a safe language (Java) [22]. Processes on JX do not share memory and run in a single hardware address space using language safety instead of hardware protection for process isolation. However, JX employs an open process architecture since, like Java, it allows dynamic code loading.

6. Conclusions and Future Work

In a quest to improve system dependability, we have defined a new sealed process architecture, which offers a number of important advantages. It creates explicit interfaces between a program and its extensions and prohibits sharing of memory. This can improve the reliability of systems in the presence of extension failures, a well-known source of failure in operating systems and applications. It allows defect detection tools and tools that verify partial program correctness to make sound assumptions about program behavior. It also increases the precision and accuracy of static analysis, which is a key component of code optimization and defect detection tools. It enables an operating system to provide stronger security guarantees than open processes and code-based process identity. It allows the elimination of redundancies between an OS and language runtimes such as the JVM and CLR. And, it encourages applications and systems programmers to practice better software engineering.

We have implemented sealed processes in the Singularity operating system. Preliminary results suggest that the restrictions imposed by sealed processes are not overly burdensome and are at least partially compensated by improved detection of coding errors. When combined with software isolated processes (SIPs), sealed processes also enable static analysis that significantly improves microbenchmark performance. It is too early to quantify the eventual impact of these improvements on macro-throughput.

We believe the sealed architecture shows sufficient promise to merit further consideration by the research community. We see two important avenues of future research. The first is implementing a larger class of extensible programs on the architecture to further evaluate its validity. For example, we believe porting a large database server would be a valuable experiment. The second is evaluating the sealed architecture in a hardware-protected operating system, such as Windows, Linux, or MINIX 3 [27].

7. References

1. Abadi, M., Birrell, A. and Wobber, T. Access Control in a World of Software Diversity. in *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa Fe, NM, 2005.
2. Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M. Mach: A New Kernel Foundation for UNIX Development. in *Summer USENIX Conference*, Atlanta, GA, 1986, 93-112.
3. Aiken, M., Fähndrich, M., Hawblitzel, C., Hunt, G. and Larus, J., Deconstructing Process Isolation. Technical Report MSR-TR-2006-43, Microsoft Research, 2005.
4. Back, G., Hsieh, W.C. and Lepreau, J. Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java. in *Proceedings of the 4th USENIX Symposium on Operating Systems Design & Implementation (OSDI)*, San Diego, CA, 2000.
5. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K. and Ustuner, A. Thorough Static Analysis of Device Drivers in *Proceedings of the EuroSys 2006 Conference*, Leuven, Belgium, 2006.
6. Barnes, F., Jacobsen, C. and Vinter, B. RMoX: A Raw-Metal occam Experiment. in *Communicating Process Architectures*, IOS Press, Enschede, the Netherlands, 2003, 269-288.
7. Bershad, B.N., Savage, S., Pardyak, P., Sirer, E.G., Fiuczynski, M., Becker, D., Eggers, S. and Chambers, C. Extensibility, Safety and Performance in the SPIN Operating System. in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, 1995, 267-284.
8. Biberstein, M., Gil, J. and Porat, S. Sealing, Encapsulation, and Mutability. in *Proceeedings of the 15th European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science*, Springer-Verlag, Budapest, Hungary, 2001.
9. Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G. and Fox, A. Microreboot—A Technique for Cheap Recovery. in *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, 2004, 31-44.
10. Chou, A., Yang, J., Chelf, B., Hallem, S. and Engler, D. An Empirical Study of Operating Systems Errors. in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Alberta, Canada, 2001, 73-88.
11. Dorward, S., Pike, R., Presotto, D.L., Ritchie, D.M., Trickey, H. and Winterbottom, P. The Inferno Operating System. *Bell Labs Technical Journal*, *2* (1). 5-18.
12. Engler, D.R., Kaashoek, M.F. and O'Toole, J., Jr. Exokernel: an Operating System Architecture for Application-Level Resource Management. in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain Resort, CO, 1995, 251-266.
13. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J.R. and Levi, S. Language Support for Fast and Reliable Message Based Communication in Singularity OS. in *To appear: EuroSys2006*, Leuven, Belgium, 2006.
14. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J.R. and Levi, S. Language Support for Fast and Reliable Message Based Communication in Singularity OS. in *Proceedings of the EuroSys 2006 Conference*, Leuven, Belgium, 2006.
15. Fähndrich, M. and Larus, J.R. Language Support for Fast and Reliable Message Based Communication in Singularity OS. in *Submitted to EuroSys2006*, 2005.
16. Fitzgerald, R., Knoblock, T.B., Ruf, E., Steensgaard, B. and Tarditi, D. Marmot: an Optimizing Compiler for Java. *Software-Practice and Experience*, *30* (3). 199-232.
17. Fitzgerald, R. and Tarditi, D. The Case for Profile-directed Selection of Garbage Collectors. in *Proceedings of the 2nd International Symposium on Memory Management (ISMM '00)*, Minneapolis, MN, 2000, 111-120.
18. Flatt, M. and Findler, R.B. Kill-safe Synchronization Abstractions. in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI 04)*, Washington, DC, 2004, 47-58.
19. Ganger, G.R., Engler, D.R., Kaashoek, M.F., Briceño, H.M., Hunt, R. and Pinckney, T. Fast and Flexible Application-level Networking on Exokernel Systems. *ACM Transactions on Computer Systems*, *20* (1). 49-83.
20. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M. and Boneh, D. Terra: A Virtual-Machine Based Platform for Trusted Computing in *Proceedings for the 19th ACM Symposium on Operating System Principles (SOSP)*, Bolton Landing, NY, 2003.
21. Goldberg, A. and Robson, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.
22. Golm, M., Felser, M., Wawersich, C. and Kleinoeder, J. The JX Operating System. in *Proceedings of the USENIX 2002 Annual Conference*, Monterey, CA, 2002, 45-58.
23. Gosling, J., Joy, B. and Steele, G. *The Java Language Specification*. Addison Wesley, 1996.
24. Härtig, H., Hohmuth, M., Liedtke, J. and Schönberg, S. The Performance of μ-kernel-based Systems. in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*, Saint Malo, France, 1997, 66-77.
25. Hawblitzel, C., Chang, C.-C., Czajkowski, G., Hu, D. and Eicken, T.v. Implementing Multiple Protection Domains in Java. in *Proceedings of the 1998 USENIX Annual Technical Conference*, New Orleans, LA, 1998, 259-270.
26. Hawblitzel, C. and Eicken, T.v. Luna: A Flexible Java Protection System. in *Proceedings of the Fifth ACM Symposium on Operating System Design and*

Implementation (OSDI '02), Boston, MA, 2002, 391-402.

27. Herder, J.N., Bos, H., Gras, B., Homburg, P. and Tanenbaum, A.S. Modula System Programming in MINIX 3. *USENIX ;login*, April, 2006.
28. Hunt, G., Larus, J., Abadi, M., Aiken, M., Barham, P., Fähndrich, M., Hawblitzel, C., Hodson, O., Levi, S., Murphy, N., Steensgaard, B., Tarditi, D., Wobber, T. and Zill, B., An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.
29. Hunt, G.C., Larus, J.R., Tarditi, D. and Wobber, T. Broad New OS Research: Challenges and Opportunities. in *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa Fe, NM, 2005.
30. Lampson, B.W. and Sproull, R.F. An Open Operating System for a Single-user Machine. in *Proceedings of the Seventh ACM Symposium on Operating Systems Principles (SOSP)*, Pacific Grove, CA, 1979, 98-105.
31. McJones, P.R. and Swar, G.F. Evolving the UNIX system interface to support multithreaded programs. in *Proceedings of the Winter 1989 USENIX Conference*, 1989.
32. Morrisett, G., Walker, D., Crary, K. and Glew, N. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, *21* (3). 527-568.
33. Murphy, B. and Levidow, B. Windows 2000 Dependability. in *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, New York, NY, 2000.
34. Paul, N. and Evans, D. NET Security: Lessons Learned and Missed from Java. in *20th Annual Computer Security Applications Conference (ACSAC)*, Tucson, AZ, 2004, 272-281.
35. Peinado, M., Chen, Y., England, P. and Manferdelli, J. NGSCB: A Trusted Open System. in *Proceedings of the 9th Australasian Conference on Information Security and Privacy (ACISP)*, Sydney, Australia, 2004.
36. Process, J.C. Application Isolation API Specification *Java Specification Request*, 2003, JSR-000121.
37. Redell, D.D., Dalal, Y.K., Horsley, T.R., Lauer, H.C., Lynch, W.C., McJones, P.R., Murray, H.G. and Purcell, S.C. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, *23* (2). 81-92.
38. Ritchie, D. and Thompson, K. The UNIX Time-Sharing System. *Communications of the ACM*, *17* (7). 365-375.
39. Rovner, P., Levin, R. and Wick, J., On Extending Modula-2 for Building Large, Integrated Systems. Technical Report SRC-3, DEC SRC, 1985.
40. Saulpaugh, T. and Mirho, C. *Inside the JavaOS Operating System*. Addison-Wesley, 1999.
41. Schroeder, M.D. and Saltzer, J.H. A Hardware Architecture for Implementing Protection Rings in *Proceedings of the Third ACM Symposium on Operating Systems Principles (SOSP)*, ACM, Palo Alto, CA, 1971.
42. Seltzer, M.I., Endo, Y., Small, C. and Smith, K.A. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. in *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI 96)*, Seattle, WA, 1996, 213-227.
43. Spear, M.F., Roeder, T., Levi, S. and Hunt, G. Solving the Starting Problem: Device Drivers as Self-Describing Artifacts. in *Proceedings of the EuroSys 2006 Conference*, Leuven, Belgium, 2006.
44. SPEC *SPECweb99 Release 1.02*. Standard Performance Evaluation Corporation Warrenton, VA, 2000.
45. Sreedhar, V.C., Burke, M. and Choi, J.-D. A Framework for Interprocedural Optimization in the Presence of Dynamic Class Loading. in *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI 00)*, Vancouver, BC, 2000, 196-207.
46. Swift, M.M., Annamalai, M., Bershad, B.N. and Levy, H.M. Recovering Device Drivers. in *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, 2004, 1-16.
47. Swift, M.M., Bershad, B.N. and Levy, H.M. Improving the Reliability of Commodity Operating Systems. in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, 2003, 207-222.
48. Swinehart, D.C., Zellweger, P.T., Beach, R.J. and Hagmann, R.B. A Structural View of the Cedar Programming Environment. *ACM Transactions on Programming Languages and Systems*, *8* (4). 419-490.
49. Thacker, C., Stewart, L.C. and Satterthwaite, E., Firefly: A multiprocessor workstation. . Technical Report SRC-023, DEC SRC, 1987.
50. Thacker, C.P. and Stewart, L.C. Firefly: a Multiprocessor Workstation. in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, 1987, 164-172.
51. von Behren, R., Condit, J., Zhou, F., Necula, G.C. and Brewer, E. Capriccio: Scalable Threads for Internet Services. in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, 2003, 268-281.
52. Vyssotsky, V.A., Corbató, F.J. and Graham, R.M. Structure of the Multics supervisor. in *AFIPS Conference Proceedings 27, 1965 Fall Joint Computing Conference (FJCC)*, Spartan Books, Washington, DC, 1965, 203-212.
53. Wahbe, R., Lucco, S., Anderson, T.E. and Graham, S.L. Efficient Software-Based Fault Isolation. in *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, Asheville, NC, 1993, 203-216.
54. Weinreb, D. and Moon, D. *Lisp Machine Manuel*. Symbolics, Inc, Cambridge, MA, 1981.