Singularity Design Note 2

Architectural Abstractions

Processes, Channel, Manifests, and Activities

This design note proposes the transient abstractions of the Singularity architectural that support system reliability—processes, channels, manifests, activities, etc. Subsequent design notes will describe the non-transient abstractions of the Singularity architecture, those abstractions which span process lifetimes. The description is broken into an introduction, overview of the Singularity system architecture, technical description of the abstractions, and discussion on the trade-offs in the design.

1. Introduction

Singularity is a new research OS that supports the construction of reliable systems. In this context a system is called reliable if it obeys some specification of its behavior[1]. Of special interest are things that the system will not do—crash, hang, corrupt data, succumb to viruses, etc.—since such specifications are often easier to state and to enforce than specifications of what the system will do, such as complete a specific task and then exit. If the specification is good enough, a reliable system never behaves in a manner unwanted or unanticipated by its designers, developers, administrators, or users. Singularity is specifically designed to enable the automatic checking of useful specifications of useful systems thereby allowing concrete statements about the reliability of Singularity systems.

Systems written in high-level languages like C# are much more practical to specify and check than systems in low-level languages like assembler and C++. Accordingly most of Singularity is written in a variant of C#, called Spec#, extended for specification. Singularity programs can be written in C#, Spec#, or other type-safe MSIL-based languages.

The lowest layer of the Singularity kernel is written in assembler and C++, which can only be checked in limited ways with primitive tools. This layer therefore contains as little code and as little functionality as possible; it consists primarily of code which cannot be expressed in MSIL, such as context switches, I/O instructions, and low-level parts of the garbage collector (GC). If the code in this layer fails, the system as a whole can fail in any way.

[1] We cannot in general check an arbitrary system against an arbitrary specification. In practice we must restrict ourselves to partial specifications, specifications covering some, but not all aspects of a systems behavior. The more complete the specifications that can be checked, the more reliable the system.

Copyright © Microsoft Corporation. All Rights Reserved.

The next layer of the Singularity kernel is written in unsafe C#, which also can only be checked in limited ways with primitive tools. This layer again contains as little code and functionality as possible; it includes most of the GC, page table, and I/O access code. If the code in this layer fails, the system can fail in almost any way.

The remainder of Singularity is written in Spec#, which is type safe and whose behavior can be specified and checked in a variety of useful ways. The code in this layer can fail only in restricted ways, and it is much more feasible to build systems that continue to operate following certain types of failure.

All third-party code, including applications, extensions, and device drivers, are presented to Singularity in the form of safe MSIL. Translation to the native instruction set can take place at install time or at load time.

Writing a system primarily in a safe MSIL-based language helps in checking it against a specification by restricting the system's possible misbehavior, but a safe language alone is not enough. Checking large, complex systems is often impossibly harder than checking small, simple systems. Strategies are needed to bound complexity and maintain checkability as systems invariably grow larger. The best way to structure a checkable system is so that the system and its specifications can be checked in small pieces, even as the system itself grows large.

Like other operating systems, Singularity provides processes to isolate pieces of the system from each other. Beyond other systems, however, Singularity exploits processes to enable checkability. Most of a Singularity system uses processes to enable checkability. Many OS services traditionally placed in the kernel are moved out into separate processes to facilitate checking for reliability.

It is hoped that the use of processes will enable the building of large Singularity systems that are nevertheless checkable.

In the future, Singularity may incorporate additional ways to isolate pieces of a system from the behavior or misbehavior of other pieces. For example, future support for transactions could help keep the system in a consistent state, and a system that uses transactions might be more practical to check than a system that does not. As new mechanisms for system reliability are provided, they will be used pervasively throughout the system.

The goal of checkability is to find problems in a system as early as possible, preferably before the system ever runs, through static analysis. Singularity extends the domain of static analysis from compile time and link time to install time and load time. When full static analysis is not possible or practical, it can of course be augmented with dynamic checking. Most problems with Singularity systems should be detected much earlier than with traditional systems.

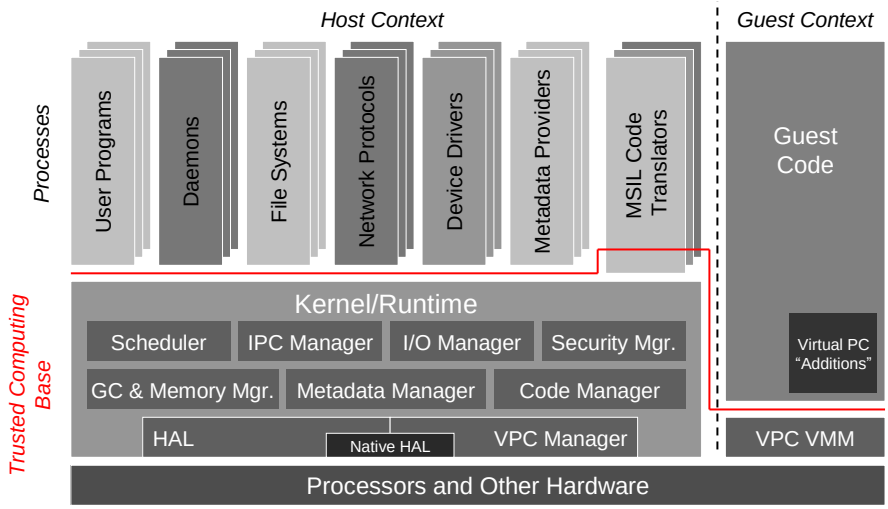
Although this discussion has focused on the design features of Singularity that support checkability for reliability, it is expected that features like type safety and process isolation will make it easier to build reliable systems in the first place. This discussion stresses checkability here because it is a novel goal for the design of an OS and its abstractions, and because one might more readily expect to quantify how practical a system is to check than how practical it is to build.

2. Singularity System Architecture

Singularity is a managed code OS. Where traditional operating systems present untyped memory and the native instruction set to application code, Singularity replaces these with the abstractions of typed memory and an abstract instruction set, in the form of type-safe MSIL. All third-party binaries,

Copyright © Microsoft Corporation. All Rights Reserved.

including applications, extensions, and device drivers, are delivered to Singularity as type-safe MSIL binaries. Abstractly, the MSIL binaries are translated into native instruction streams at load time. In practice, native instruction streams can often be generated at load time based on application metadata. Caching of native instruction streams is invisible to both applications and users.



The Singularity trusted computing base (TCB) maintains four fundamental security policies[2]. First, the TCB insures the integrity and isolation of all address spaces. Second, the TCB insures the integrity and isolation of all processes. Third, the TCB insures the integrity of interprocess communication (IPC). Fourth, the TCB insures that a process can verify its own identity and the identity of any processes with which it directly communicates.

The TCB insures its security policies by guaranteeing that no untrusted or unverified native instructions are ever executed except inside the secured Virtual PC Guest Context. The rest of the system is protected from code running in the Virtual PC Guest Context because it runs in its own address space which is contained through hardware memory protection. The TCB guarantees the integrity of processes by insuring that no process has direct access to the memory of any other process.

No third party code is allowed to run within the Singularity TCB. Where other operating systems might load third party code into the kernel in the form of device drivers, Singularity hosts device drivers outside the kernel in processes.

The Singularity kernel integrates most of the services of the Common Language Runtime (CLR) with traditional kernel-based OS services such as scheduling, IPC, and I/O management. By combining the runtime into the kernel, Singularity eliminates redundancies in resource management and security policy. The runtime also enjoys access to kernel features, such as direct access to processor memory management units.

[2] The TCB for many applications will include components outside the Singularity TCB. For example, many applications will rely on additional security policies, such as the security of the file system. For those programs, the file system is a necessary component of their TCB.

Copyright © Microsoft Corporation. All Rights Reserved.

In Singularity, the CLR is factored to minimize code in the trusted computing base. Singularity factors the CLR into five key components: code translators, the code manager, metadata providers, the metadata manager, and the memory manager. Code translators and metadata providers reside outside the kernel in processes. Code translators convert MSIL into verifiable native instruction streams. The code manager verifies and caches native instruction streams and handles their mapping into processes. Metadata providers store metadata for in-process reflection and introspection for cross-process analysis. The metadata manager connects metadata providers to the memory system, code translators, the code manager, and the memory manager. The memory manager includes the GC and its accompanying facilitates such as the GC write barrier.

Singularity supports multiple code translators. Individual code translators generate qualitatively different code from the same MSIL input. For example, translators may make differing tradeoffs between performance and security or target secondary processors such a GPU or a network offload engine. Code translators reside outside the trusted computing base. Code translators either provide proof carrying code or verifiable native instruction streams so that the code manager in the kernel can confirm that the native instruction stream maintains type safety before execution. Cached native instruction streams may be signed after verification by the code manager to prevent tampering.

3. Technical Description

3.1. Processes, Messages, and Channels

A **process** is the basic container of computation and unit of isolation in Singularity. A process defines an object space that circumscribes the objects of the computation. The object space of a process is disjoint from all other processes. No CLR objects are shared between Singularity processes and all CLR objects belong to exactly one process.

The computation within a process communicates with the external entities exclusively by sending **messages** to other processes. Messages are copied from a sending process to a receiving process through a two-way asynchronous communication conduit called a **channel**. A channel has exactly two endpoints.

Channels are strongly typed by a **channel contract**. The channel contract defines the relationship between the two ends of a channel, the **channel endpoints**. The channel contract defines a set of message that may be sent through the channel and a pattern describing allowable sequences of message exchange through the channel. Directly derivable from the channel contract is the set of endpoint states and the allowable transitions between those states in terms of messages sent or received. The channel contract definition is sufficiently explicit that code implementing either end of a channel contract may be statically checked for conformance to the contract.

There is no sharing of data between processes in Singularity. All communication is either via copy or hand off of memory; while the two have different performance characteristics, both maintain a key invariant: at any given instance a piece of data is owned exclusively by one process. For example, device memory may be mapped into a process, but is not shared between processes. Similarly, data buffers in a networking stack might move from layer to layer, but only one process has access rights to the buffer at any moment.

Messages sent through channels may contain only values and channel endpoints. Object references cannot be sent between processes since each process has its own object space, objects must be marshaled by value to be passed in messages between processes.

Copyright © Microsoft Corporation. All Rights Reserved.

3.2. Channel Endpoints

The terminus of a channel within a process is a channel endpoint. When the kernel creates a channel, it returns two typed channel endpoints to the requester. The requester can either move both endpoints to other processes through messages, or in the more common case, it can retain one endpoint and move the other endpoint to another process through a message.

The channel endpoint is exposed to code within a process through an instance of a **channel endpoint object** that implements CLR interfaces specific to the channel type. A process contains one channel endpoint object for each channel endpoint it accesses. By making calls on the endpoint object, process code invokes the kernel message transport mechanism for the corresponding channel. Singularity will provide "syntactic sugar" to express alternative communication patterns to the same channel type. For example, Singularity will provide both RPC and asynchronous access to the same channel type, possibly through different but related interfaces on the endpoint object, both interfaces derived from the same channel type specification.

Channel endpoints can be moved to other processes through messages. However, a channel endpoint cannot be moved once it has been used to either send or receive a message. This property guarantees that any process receiving a channel endpoint always receives the endpoint in a known state.

To move a channel endpoint to another process, the sending process places a reference to the endpoint object in a message. The kernel marshals the channel endpoint encapsulated by the endpoint object from the sending to the receiving process. On the receiving side, a new endpoint object is created to encapsulate the channel endpoint. While the sending code may retain a reference to the endpoint object for a now-departed endpoint, the sending process no longer has access to the departed endpoint; any attempt to invoke a send or receive method on the endpoint object will produce an error as the endpoint object is no longer bound to a channel.

A channel endpoint may be thought of as a capability to send messages on the channel. Processes cannot create their own channels, but must instead receive them from the outside, such as from the kernel. A process can refer to a channel only through a channel endpoint.

3.3. Address Spaces

It is important to note that unlike processes in most other operating systems, each Singularity process need not have its own unique **address space** (a linear mapping of memory addresses to memory locations). The creation of an address space in Singularity is a deployment decision independent of the creation of a process. Because the MSIL code for a Singularity process can refer to objects only by object references, not by addresses, Singularity is not constrained to a one-to-one relationship between processes and address spaces. While the expected number of processes in Singularity is much higher than in a traditional OS, the expected number of address spaces is much lower. The translation of object references to memory addresses is invisible to Singularity processes.

3.4. Manifests

The code of a Singularity process is statically defined at the time of process creation. The process creation mechanism incorporates an abstraction, called a **manifest**, which itemizes all code to be included in the process. Once a process has been created, no new code may be introduced into the process. In more concrete terms, Singularity has no `LoadLibrary` or `LoadClass` APIs because Singularity does not allow the dynamic insertion of new code into an existing process. Singularity

Copyright © Microsoft Corporation. All Rights Reserved.

supports dynamically linked libraries (DLLs), as a mechanism to enable independent static servicing of code, but to be loaded into a process the DLLs must be included in the process at creation time.

Singularity disallows the loading of new code into an existing process. The only mechanism in Singularity for extending the functionality of an existing process is through communication with another process. Where applications on other operating systems call a `LoadLibrary` or `LoadClass` API to acquire new functionality, Singularity applications create a child process to hold the extension code.

A process manifest defines the static contents of a process. It includes not just the code, but a set of statically defined channel endpoints used to bind a process to its environment and the types of all channel endpoints allowed in the process. As with code, the set of channel types is fixed at the time of process creation and cannot change. Note that the manifest defines the types, but doesn't limit the number of channels; this is very analogous to the way that code defines the types of objects that can be instantiated but does not limit the quantity of instantiation.

Before a process begins execution, its statically defined channel endpoints objects are seeded with an initial set of channel endpoints through which it accesses the kernel and other processes. One of the endpoints defines the address space into which the process is to be instantiated. This initial set of channel endpoints must include a channel through which the process can request additional services and through which the process is notified of important events. In the simplest case, the initial channel endpoints are direct channels to the kernel. However, the channel endpoints can connect to other processes, such as when a process interposes itself between the kernel and the new process.

3.5. Kernel and OS Services

All kernel abstractions and OS services are exposed through channel endpoints. Within a process, each channel endpoint is visible through a unique CLR endpoint object. The implementations of methods on the endpoint object are effectively stubs which communicate through the channel to the kernel (or other process) where the real implementation of the abstraction resides.

The instantiation of a new kernel abstraction, such as the creation of a new process, always results in the creation of a channel used to communicate with the code (in the kernel) implementing the abstraction.

3.6. Threads

A **thread** describes an execution context within a process. At a minimum, a thread is defined by a set of registers, including a program counter. While programs are allowed to create, suspend, and resume threads, Singularity strictly controls the allowed state of all threads. In particular, Singularity forbids out-of-band modifications of thread state that could result in the violation of memory type safety.

3.7. Activities and Reservations

An **activity** is an abstraction in Singularity for managing resources across and within processes. Associated with an activity is a set of reservations. A **reservation** is an abstraction for pre-arranging the scheduling of resources over a period of time.

Similar to address spaces, activities with their accompanying reservations are not necessarily bound to a single process. Unlike an address space, a process may own multiple activities. Processes cooperating to achieve a common task will often share an activity. Moreover, a single process might incorporate multiple activities to represent differing resource needs.

Copyright © Microsoft Corporation. All Rights Reserved.

Activities can be bound to the two schedulable entities within Singularity: threads and messages. Binding an activity to a thread tells the scheduler to apply the resources of the activity to the thread. Binding an activity to a message tells the scheduler to bind the activity to the thread in which the message is serviced.

Where more than one thread is bound to an activity, the resources of the activity are shared between the threads. As well as receiving their reserved resources, activities also receive a proportionate share of any unreserved resources.

3.8. Lifetime Management and Process Tree

Processes are the primary unit of lifetime management in Singularity. The lifetime of all other system abstractions are manipulated through the lifetimes of processes. The lifetimes of threads, channel endpoints, address spaces, and activities are maintained through reference counts; each is destroyed when it is no longer associated with any non-terminated process. The lifetimes of channels are maintained indirectly through ownership on their associate endpoints and the lifetimes of reservations are maintained indirectly through ownership on their associated activities.

When the last process holding a reference count to a thread, channel endpoint, channel, address space, or activity is terminated, the referenced abstraction is immediately destroyed by the Singularity kernel. As mentioned previously, threads and channel endpoints have at most one owner. If all of the channels referenced by a process have been closed, the process may be terminated at any time by the kernel because it has no positive ability to affect the causal future of the system.

Within a process, a thread can terminate itself only when it has completely unwound its activation stack. By forcing threads to completely unwind their stack before terminating, Singularity enables static verification that a thread exiting a process has maintained all object invariants. When a process terminates, all threads in the process are immediately destroyed without unwinding their stack because in-process invariants are no longer applicable.

When one endpoint of a channel is destroyed, the channel is closed and the process holding the remaining endpoint is notified; any attempts to send or receive messages on a closed channel abort in failure.

Processes reside in a strict process tree. Each process has exactly one parent in the tree and the Singularity kernel is the root of the process tree. Even after a parent process has terminated, it continues to exist, but does not hold resources, until all of its children have been destroyed; a child process is not terminated by the termination of its parent. A process with no children is immediately destroyed on termination. By maintaining its position in the process tree even after termination, a parent process retains its role as a reference point for metadata and can be used to control its offspring. For example, with appropriate authority, a process can ask the kernel to terminate all of the processes in the process subtree of a terminated process.

The position of an executing or terminated process in the process tree cannot be changed. However, a process can be moved in the process tree after it has been created, but before it begins execution. For example, a shell might start a background process by creating the process, partially binding its statically defined endpoints to a set of channels, such as channels for `stdin` and `stdout`, and then asking another process to become its owner. The process is grafted into the process tree of the new owner. The new owner would start the process, possibly after binding other statically defined endpoints such as `stderr`. Similar to the movement rules on channel endpoints, once a process has

Copyright © Microsoft Corporation. All Rights Reserved.

begun execution, neither can it be relocated in the process tree nor can its statically defined channel endpoints be modified.

4. Discussion

While Singularity does not differ dramatically from an idealized microkernel OS, a number of its features are unique, particularly in the choice of design tradeoffs. Singularity's design is influenced by a strong affinity for design simplicity and tools-based analysis over performance or feature richness.

A guiding philosophy behind Singularity is one of simplicity over richness. Wherever possible, the set of design space choices available to the programmer is constrained to facilitate reasoning about the resulting software artifact. Singularity especially favors restrictions to the programming model when such restrictions result in a dramatic increase in the ability of tools to detect sources of failure early in the life of the system or application. For example, Singularity demands that all code outside the kernel must be expressed in type-safe MSIL. This absolute restriction limits the code that can be written to Singularity, but enables Singularity to verify type safety in all programs before they are allowed to run.

4.1. Reliability

Just as a trusted computing base is defined by the set of security objectives it tries to maintain, the reliability of a system can be defined by a set of reliability objectives. System reliability can be affected by hardware failures, programmer failures, and user failures. The primary objective of the abstractions described in this design note is to maximize system reliability against programmer failures.

Singularity maximizes system reliability against programmer failure in two key ways. First, Singularity maximizes the ability of tools-based analysis to catch programmer errors as early as possible. Second, Singularity minimizes the possibility that a programmer-caused failure in one system component will result in a cascade of failures in other components.

4.2. Processes

The Singularity process model places a premium on strong isolation and explicit communication in order to maximize the impact of tools-based analysis to detect programmer errors.

One way programmers minimize error is by defining and maintaining object invariants. Object invariants typically define a set of allowed object states and the conditions under which the invariants will be maintained. A typical object method will assert that the object's invariants hold, and then modify the state of the object under protection of a lock or other mechanism; finally, the method insure that the object's invariants have been re-established before leaving the protected region of code.

The Singularity process model defines an isolated object space for each process to help maintain object invariants. As long as a process has not been terminated, Singularity guarantees that the object code that maintains its invariants will always run to completion. When a process terminates, Singularity guarantees that no other process will see any inconsistent objects. Singularity achieves these guarantees by forcing the collocation of all threads that can directly alter an object's state to reside within the same process.

The Singularity process model requires more discipline of programmers, but improves the chances that a failure in one component will not result in a cascade of unexpected failures in other components.

Copyright © Microsoft Corporation. All Rights Reserved.

While the model requires additional programmer effort between cooperating processes and has performance implications, it greatly reduces the types of side effects that must be considered by both programmer and tools.

Sealing the code of a Singularity process at the time of creation encourages programmers to use a development model where code from third parties is loaded into a separate process. Loading third party extension code into a separate process increases isolation and reduces the unintended propagation of failures. System reliability is increased by eliminating unintended sharing. In systems where extension is achieved through dynamic loading of code, unintended sharing is a common occurrence. In the typical case, a host defines some partially opaque shared interface either in code or data. The developer of the extension then writes code which walks through the interface to access data or code not intended to be shared. Unintended sharing typically results in a proliferation of unspecified code dependencies. The process-based extension model of Singularity eliminates the occurrence of unintended sharing because all sharing is explicit.

Process isolation eliminates the propagation of memory corruption from errant code. In Singularity, an errant parent or child cannot corrupt the memory of its relation. A child might convince a parent to corrupt its own memory, but this is only possible through faults in the parent's communication code. Because the communication between parent and child takes place through well defined communication channels, static and dynamic checks can be applied to insure that both parent and child abide by the contract of the channel.

Process isolation improves program reliability by decoupling the lifetime of parent and child processes. In systems where extension code is loaded into the host process, the extension has no ability to reach a clean state in the event that the host process is terminated forcefully. In Singularity, a child process can gracefully terminate itself when it is informed of its parent's death through notifications on any shared channels.

4.3. Communication

At first glance, the design of Singularity's communication channel abstraction may seem arbitrary. For example, one could easily argue that in theory two one-way channels are equivalent or perhaps even more flexible than one two-way channel. However, tools-based verification of conformance to a protocol is drastically simpler when all of the messages for a protocol transit through a single two-way channel.

While channel endpoints can be moved in messages between two processes, Singularity demands that a channel endpoint can be moved only in its pristine state, i.e. when no messages have been sent or received through the endpoint. This property, called the bound output restriction in pi-calculus, insures that a process always receives a channel in a known state and drastically simplifies conformance verification and broadens its reach. Similar benefits are derived from the corollary that ownership of a process must be transferred before the process begins execution.

There is no simultaneous sharing of data between Singularity processes. While Singularity may make optimizations such as sharing of message data between processes to minimize memory footprint, the implementation of these optimizations must not be visible to either the sending or receiving process. By forbidding sharing through memory, Singularity restricts communication between processes to explicit communication through channels, not implicit communication through shared memory. This restriction helps enforce verifiable interfaces between processes and maintain a strong isolation model.

Copyright © Microsoft Corporation. All Rights Reserved.

4.4. Performance

Singularity is intended to provide "good enough" performance, but no better. The primary goal of Singularity is to increase system and application reliability. The belief is that users have enough performance; they need and want more reliable systems, systems worthy of their trust.

The restrictions Singularity places on the programming model actually result in a number of interesting opportunities for optimization underneath the applications where such optimizations do not compromise the quality of application code. For example, Singularity can make very concrete statements about the usage of dynamic types within a process at load time whereas systems with `LoadClass` or `LoadLibrary` APIs would be forced to make very conservative assumptions.

Singularity can make a number of optimizations that reduce the costs of strong isolation by exploiting the fact that all code running outside the Singularity kernel is expressed using type-safe MSIL. By hosting multiple processes within a shared address space context switches between co-located processes need not flush TLB or cache entries. Furthermore it is expected that in many Singularity deployment scenarios, the kernel and all other processes will share a single address space, thus removing a primary cost of context switches across the entire system.

Copyright © Microsoft Corporation. All Rights Reserved.