Singularity Design Note 4

Process Model

Threads, Execution, and Communication in Processes

This design note describes the contents inside a process. It describes Singularity's model for execution and the manifestation of channel endpoints and messages as accessed by process code.

1. Introduction

Singularity Design Note 2 (SDN2) describes the basic process-based architecture of Singularity including the existence of processes, channels, channel endpoints, address spaces, activities, reservations, and threads. However, SDN2 does not describe how these abstractions are surfaced within a process.

This design note describes the process model for Singularity. It describes how the abstractions enumerated in SDN2 are realized and made visible to code running within a Singularity process. This design note places particular emphasis on how threads, messages, and channels are made visible to code within a Singularity process.

2. Building Blocks

Singularity is a managed code operating system. With the exception of the lowest layers of the kernel and code required to support the Virtual PC (VPC), all code in a Singularity system is expressed in type safe MSIL. With the exception of processes hosting VPC guest systems, all Singularity processes are generated entirely from managed code.

2.1. Address Spaces and Hardware Privilege Levels

Singularity leverages the type safety of managed code to eliminate unnecessary hardware protection boundaries between the kernel and processes. Unless explicitly configured otherwise, Singularity processes run at the same hardware privilege level as the kernel. In many cases all Singularity processes run in a single address space so that a transition from user code to kernel code need not involve a change of virtual address space. The relaxation of these hardware protection boundaries is possible because Singularity has complete control over the native instruction stream presented to system processors. Managed user code can express object references in the process object space, but cannot express addresses in either the virtual or physical address spaces. Managed code is unaware of the address space in which it runs.

Singularity uses memory management hardware for two purposes: to create distinct virtual address spaces and to isolate processes that contain code which is not verifiably type safe. Support for non-verified code is necessary for hosting Virtual PC code and for testing of untrusted code generators. Distinct address spaces are typically required when an application or set of applications, like a database server, use excessive amounts of virtual address space. In these cases, memory management hardware is used not for protection, but to provide a unique virtual address space.

2.2. Kernel Code

In Singularity, kernel code includes all code which is trusted with the capability to bypass process or address space isolation boundaries. By this definition, kernel code includes not only the privileged code explicitly running in the kernel proper, but also code exported by the kernel into user processes.

The implementation of the boundary between user and kernel code in Singularity differs in implementation, but not intention, from traditional operating systems which rely on hardware protection mechanisms. For example, in a hardware-protected OS, the transition from user to kernel code is typically demarcated by a trap. The instruction stream before the trap cannot be protected and therefore is not trusted while the stream after the trap is protected by the hardware and trusted.

In Singularity the transition from user to kernel code is often be marked by nothing more than a call a privileged subroutine. Such a transition is possible because managed user code is unable to access or subvert the instruction stream of kernel code running within the user process. With of a trusted code generator, the kernel code may even be inlined into the native instruction stream of managed user code.

The kernel proper consists of a number of code and data pages mapped into a protected region of each address space. In address spaces consisting solely of verifiably type-safe code, kernel code and data are protected by type safety. In address spaces containing untrusted instruction streams, kernel code and data pages are guarded by the hardware page protection.

2.3. Processes

From the perspective of user code running within a process, the process is an object space composed of objects reachable via object references. Managed user code is incapable of creating object references and relies on trusted kernel code to allocate objects and return object references. In Singularity the language runtime is a trusted component of the kernel

From the perspective of kernel code, a process consists of a collection of memory pages, threads, and resource references with a shared lifetime. Most pages in a process are owned exclusively, such as pages containing thread stacks or process objects. Other pages such as code pages and data pages used to communicate with other processes may be shared. When pages are shared, Singularity maintains the process invariant the sharing is not visible to user code. Pages owned by the kernel, such as channel endpoint tables, may be associated with the process for easy removal on process termination.

Processes are manipulated externally via the process abstraction exposed through channels from the kernel. Multiple channels may exist to the same process abstraction; for example, a shell and parent process may both have channels to the process abstraction for a child process. Code that implements the process abstraction resides entirely within the kernel proper.

2.4. Primitive Operations

The Singularity kernel exposes a small set of primitive operations into every process. Primitive operations are either sufficiently elemental that they must either exist within a process in order to enable channel communication or they make no sense across channel boundaries. Primitive operations are not channels to processes; they are direct entry points into kernel code and do not use messages.

Singularity divides the OS APIs into primitive operations and channel-based operations. All operations that can causally affect the state of other processes are channel based. Operations which manipulate state relevant solely to the calling process are exposed as primitive operations. For example, all thread synchronization operations are primitives operations because their causal effect is isolated to the calling process. On the other hand, APIs for manipulating activities are channel based because they causally influence other processes. The only exceptions, of necessity, to this design principle are the operations for sending and receiving messages on channels, which must be exposed as primitive operations.

Singularity primitive operations include the following:

- Thread construction and destruction operations.
- Channel construction and destruction operations.
- Operations on thread synchronization primitives such as mutexs and condition variables. These operations include construction and destruction and synchronization operations like acquire and release on a mutex.
- Endpoint bind and unbind operations.
- Message send and receive operations.
- Operations to determine the security principal associated with the opposite endpoint of a channel.

2.5. Channels

As describes in Singularity Design Note 2, a Singularity channel is a bi-direction communication mechanism for transporting messages between channel endpoints.

2.5.1. Contracts

Communication over a Singularity channel is constrained by a communication contract. The contract specifies a set of messages and a pattern describing allowable sequences of message exchange.

The following listings contain the contract for a name server and the contract required of any service registering with the name server.

Contract for a name server:

```
Contract NameServer {
    Input Message Register(String name, Chan <NamedService> NameServiceChannel);
    Input Message Lookup<C>(String name, Contract<C,S0> k);

    Output Message RegisterAck;
```



```
    Output Message RegisterNack;
    Output Message LookupAck <C>(Chan<C,S0> c);
    Output Message LookupNack;

    START: Register ?  -> (REGSISTER-ACK #  REGISTER-NACK)
              + Lookup<C>? -> (LOOKUP-ACK<C> # LOOKUP-NACK)
    REGISTER-ACK:        RegisterAck!. START
    REGISTER-NACK:       RegisterNack!. START
    LOOKUP-ACK<C>:       LookupAck<C>! . START
    LOOKUPNACK:          LookupNack!. START
}
```

Contract for a service registering with a name service:

```
Contract NamedService {
    Input Message NewClient<C>(Contract<C,S0> k);

    Output Message NewClientNack;
    Output Message NewClientAck<C>(Chan<C,S0> c);

    START:          NewClient<C> ?  -> (ACK<C> #  NACK)
    ACK<C>:         NewClientAck<C> ! . START
    NACK:           NewClientNack!. START
```

For more information on channel contracts see Singularity Design Note 5.

2.5.2. Channel Construction

Abstractly, a channel consists of two strict FIFO queues, one receiving queue for each channel endpoint. Messages within a single channel are always received in the same order as sent, and the receiving thread has access to only the first message in the queue. Between channels Singularity does not make delivery guarantees; two messages sent on separate channels may be received in a different order than they were sent.

Communication through a channel is asynchronous, send and receive operations are decoupled. Send operations on channel endpoints can be thought of as non-blocking operations. While the scheduler may take note of a send operation to alter thread scheduling, it should be noted that the scheduler is free at any time to reschedule threads as needed to meet scheduling needs.

Receive operations on a channel endpoint block if no message is available. The process primitive operations for receive support timeout and polling variants. Singularity also exposes process primitive operations for receiving from multiple channels.

The sizes of a channel's queues are bounded only by available system memory and by their channel contract. Channel queues can be bounded by their contract when the contract state machine constrains the maximum number of possible outstanding messages. Once all system memory has been exhausted, or all system memory available to a process as governed by memory quotas, any attempt to send an additional message results in an error and the immediate closure of the channel.

When a channel is closed through either of its channel endpoints, all messages in the channel queues are immediate discarded and any further attempt to send or receive on the channel's endpoints results in an error. The holder of the opposite channel endpoint is notified that the channel has been closed.

2.5.3. Channel Ownership

At any given instance, a channel endpoint is owned by exactly one thread. Only the owning thread can legally send or receive messages through the endpoint. Ownership of a channel endpoint can be transfer either by placing the endpoint in a message or by passing it through shared memory within a process.

In order to transfer a channel endpoint between threads, both the sending and receiving threads must agree on the exact state of the channel endpoint within the channel contract state machine. The contract conformation verification system requires that channels can be exchange only when they are in an unambiguous state. The state of the endpoint can be statically defined within the type system as a 2-part type specifying the contract and state within that contract, `<contract,state>`.

Once a channel endpoint is placed in a message and the message is place into a second channel endpoint, ownership of the endpoint is immediately transferred to the receiving thread. When passing channel endpoints through shared memory within a process, ownership can be transferred in either of two ways. Either ownership is explicitly transferred to a named thread if known, or the endpoint is marked as transferable (unbound), but ownership is not actually transferred until another thread issue a bind operation on the endpoint.

2.5.4. Scheduling

The precise transfer of endpoint ownership is extremely important in meeting Singularity's soft real time guarantees. As described in Singularity Design Note 2, an activity can be attached to a message indicating that the activity's resources should be made available to the receiving thread. Because a channel endpoint is always owned by exactly one thread, the receiving thread for activity resources is always known. This clarity is particular important in scenarios where the scheduler must transfer thread priorities to avoid priority inversions.

2.6. Messages and Representations

A Singularity **message** is the basic element of communication passed through a channel between two processes. Messages are created in special heaps called communications heaps. Communications heaps consist exclusively of data structures, called **representations**, which are process neutral and can be moved between processes. Each message consists of one or more representations. Only representations contains in communications heaps can be passed between processes.

A message is sent between processes by handing off ownership of the representations contained in the message from the source process to the destination process. A message send with copy semantics is implemented through one or more representation allocations followed immediately by copy of the message contents into the representations and a representation handoff.

A representation defines a process neutral data layout. A representation defines fields, but not methods. Processes operate on representation by casting them into objects or structs which derive their shape from the representation.

From an implementation perspective, representations share many properties with CLR value types (C# structs). Representations can be thought of unboxed types. To access a representation using an abstract type, the representation must be boxed. Boxing a representation converts it into a first class CLR object by, at a minimum, associating it with a vtable. As a compiler optimization, when code accessing a representation does not access the representation through an abstract type the code can directly use the representation without boxing it.

As another implementation optimization, a boxed representation can be accessed through a surrogate object. The surrogate object contains the object's vtable, a pointer to the representation, and any additional fields used in the object, but not contained in the representation. The primary advantage of a surrogate object is that representations need not be marshaled or unmarshaled as they move between processes.

3. Discussion

3.1. Channel Model

The channel and threading semantics in Singularity were chosen by considering the following criteria (in order):

- Static checkability.
- Dynamic checkability.
- Richness of expression.
- Ease of understanding programs.
- Ease of writing programs.
- Performance.

A number of alternative channel proposals were considered, including unbuffered uni-directional channels. While these alternatives offered advantages in terms of minimalism, they limited the ability of single-channel contracts to describe the interaction between communicating threads. Other proposals of merit would have required conformance checking substantially beyond what is currently understood.

The current channel design stresses richness of description while maintaining the paramount requirement that channel contracts and channel implementers be statically checkable for conformance. The current design is particularly appealing as it drastically simplifies analysis for conformance to channel contracts. It does not require conformance checking beyond what is already quite well understood.

3.2. Cost Assumptions

This current design requires that threads and channel be quite inexpensive. It is expected that threads can be much cheaper in Singularity than in many other operating systems for two primary reasons. First, Singularity can exploit control over the instruction stream generated from MSIL to inline stack management code to minimize the cost of thread stacks. Second, the cost of a thread context switch in Singularity should approach the cost of a thread context in traditional user-mode thread implementations particular. Two factors should reduce the cost of context switch in Singularity: the ability to run user code within the same hardware protection domain as the OS kernel and the ability to run user code within the same address space as the OS kernel.

