Singularity Design Note

3

Design Groups

Design Group Charters

The Singularity project is divided into roughly fifteen design groups. Each design group focuses on a research topic or portion of Singularity. This document contains the charters of each design group in alphabetical order.

The Singularity website contains a list of group owners and group members. Group members are divided into three classes: owners, contributors, and interested parties.

The owner of each group has responsibility for driving forward progress. The owner makes sure design decisions are made in a timely manner. Owners should strive for buy-in on decisions, but consensus is not an absolute requirement.

Contributors in each group are responsible for contributing to forward progress. Contributors should make sure that reasonable options are explored, but should also recognize the importance of making forward progress. Singularity will evolve through multiple prototypes; neglected areas of the design space can be revisited in future prototypes. Owners can expect contributors to take regular design assignments, such as writing text for a design note, etc.

Interest parties are people who expressed an interest in tracking progress in an area, but not actively contributing. They can be listed as optional attendees at meetings. They can also track progress via meeting notes sent to singnote. Owners can expect interested parties to take periodic design assignments, such as reading a design note and providing feedback in a reasonable period, etc.

1. Application Abstraction

The charter of the Singularity Application Abstraction Design Group is to design the application abstraction, design its representation in the application manifest, determine how the application abstraction gets mapped into the metadata subsystem, and ultimately provide the implementation to maintain the connection between application abstraction, application instance, and processes.

The Singularity Application abstraction is a mechanism for describing a program to Singularity. An application names the binaries and other files, settings, and channel endpoints that combined create a program. An application is embodied in an application manifest. Like a .exe file is used to create a process in Windows, an application manifest is used to create a process (or process tree) in Singularity.

All running processes in Singularity belong to some application instance. All executable code in a Singularity disk image belongs to some application. Even the Singularity kernel is described by an application abstraction. If a binary on a Singularity disk image isn't named by an application abstraction, then it isn't executable.

Singularity maintains the metadata connection between an application and its application instances and their processes.

A new application abstraction can be created that describes an application composed of other applications. For example, the MSRN Streaming Media application might include some custom component code along with the Windows Media Player and PowerPoint applications as components.

Singularity applications are protected and regulated entities. The creation of new applications is carefully regulated by Singularity. Singularity enforces policies on application creation, such as requiring explicit user authorization before adding a new application or controlling the binding of an application to protected names. To improve system reliability, Singularity also protects the constituents of an application so that, for example, a user can't accidentally delete a critical piece of an application.

2. Build and Check-in Process

The Singularity Build and Check-in Process Group's charter is to put tools, procedures, and conventions into place to try to ensure that:

- What someone else can build and run, you can build and run, both now and in the future, with the same results.
- When people check in code, after the check-in the functionality that worked before still works.
- When people check in code, the code changes are announced and described to other project members.
- Coding standards allow people to edit one another's code in a seamless fashion
- Developers can use state-of-the-art development tools.

Our scope includes both the development process for builders of the Singularity system and the development and usage environments for users of Singularity.

3. Code Translation

The code translator is a component that provides translation from a representation like MSIL to native machine code. Translation is a primitive of the Singularity operating system and must be trusted. To reduce the size of the trusted computing base, the team will lead the effort to develop a type system and a type checker for typed assembly language. Because people outside the Bartok compiler team may want to augment or modify the code translator, a process will be established for proposing and implementing changes. To improve the performance of Singularity and applications running on Singularity, the team will investigate cross-process and hot-path optimizations. In addition, the team will investigate optimizations that improve the performance of systems code written in safe languages. In the long term, the code translator will provide translation to multiple architectures, including

computing cores other than CPUs. It is envisioned that compilation occurs at install time and that a just-in-time compiler is not needed.

4. Execution Environment

Current, managed execution environments (EEs) such as the CLR or the JVM, are complex, intertwined, and informally and incompletely specified. Furthermore, mechanisms are not available to define relationships between EEs, which results in large monolithic designs. Our goal is to bring greater rigor to defining execution environments so that their properties are easier to understand and check. We envision EEs being defined for specific application domains and EE specifications being defined in relation to other EEs, just as modules are partly defined by their relationship to other modules.

To this end, we will develop an approach to EE design and description that allows multiple EE's to be described and their relationships documented. The Singularity system gives us a unique opportunity to rethink the relationship between the OS, the EE, the application, and other software domains, such as device drivers. In our vision, each of these domains executes only in the context of a specific EE.

A top priority for the group is to write a concrete specification of a minimal EE that forms the basis for the execution of the Singularity kernel. Making the kernel EE as small as possible decreases the size of the trusted code base of the system, and can be used to form the basis for other EE descriptions layered on top of the kernel, such as the application EE.

Our specifications will contain both formal and informal parts. For example, the APIs of the class libraries available in each EE provide a formal, checkable specification for what functionality the EE supports. We intend to explore the extent to which an EE specification can be formalized, and plan to work with other design groups to develop tools that automate EE checking. Specifically, we anticipate checking an EE implementation against a EE specification, as well as checking a application.

We will also develop other EE specifications as time allows. As a most general case, the application-level EE (the equivalent of the current CLR), needs to be defined. In addition, we will consider defining EEs for more constrained contexts, which might include device drivers, stored-procedures, application plug-ins, etc. An important goal for domain-specific EE design is to specify a minimal environment which is required to host a domain of applications. These specifications will give us insight about the effectiveness of our approach in terms of expressiveness (how easy is it to express a given domain) and factoring (how much of an existing specification can we used to express a new one).

5. GC and Memory Management

The goal is to design and build an efficient, VM-aware GC infrastructure that has soft real time guarantees, and can be leveraged to improve application reliability and efficiency. Multiple GCs—to the extent that they are needed to satisfy these goals--will be provided, with at least one a soft-real time collector that must collaborate with the Singularity scheduler to provide maximum pause times in the range of a low number of milliseconds. At this point, there is little interest in building concurrent or parallel GCs that scale to 64-128 processor cores.

6. I/O Subsystem[1]

The group is responsible for the Singularity I/O System, File Systems and Network Stacks. It will design, implement and evaluate mechanisms and abstractions for I/O within Singularity. These include unified driver models and application interfaces for network and traditional I/O; composition and interposition of drivers and protocols; a system-wide buffer abstraction which uses metadata to expose the provenance of data and properties of communication channels; pervasive support for resource partitioning and scheduling; and an I/O API supporting concurrency, reordering and asynchronous notifications.

The I/O System in Singularity is a framework which orchestrates the movement of bulk data between hardware, device drivers, network protocols, file systems and applications. It is responsible for sharing system resources including data buffers, I/O bandwidth and storage capacity in an efficient, scheduled and resource partitioned manner so as to enable soft-real-time applications and prevent live-lock, denial-of-service, and other common attacks. It will provide a notification mechanism capable of dealing with both dynamic changes in hardware ("Plug & Play") and transient network connectivity, with an eye towards treating plug and play devices as transient network connections, with universal discovery mechanisms.

Device drivers are implemented entirely in managed code and communicate solely using Singularity channels and typed messages, with the exception of device registers that are mapped directly into drivers in a strongly typed way. Data buffers are never shared and are passed between driver processes as though copied or moved atomically. The I/O system exploits the GC, memory and type- systems wherever possible to provide semantics-preserving optimizations such as zero-copy 'under-the-hood'. Time dependencies, complex register semantics, and the need for temporarily blocking interrupts will be described as part of the driver code, with the specification checked for real-time requirements and safe execution—this may require specific language support.

Data buffers may be tagged with metadata which provides additional information about the provenance of the data they contain and the path it has taken, e.g. network data may be tagged with security of the connection, packet loss, round-trip latency, signal strength and receiving interface. Metadata may also be exploited in the context of file I/O to more effectively support undo, replication, synchronization, reconciliation, etc. as well as the diverse semantics of storage media and file systems, all the while making sure the I/O system will perform well, and to the extent meaningful for bulk data, which is likely to be untyped at the lower levels of the networks stack.

The API exposed by the I/O System will provide explicit support for I/O concurrency, partial-ordering of requests and asynchronous notification of completions and other important events. Integration with functional, declarative and concurrent programming languages will be investigated.

7. Kernel

[1] The I/O group currently includes the File System and Networking groups. It is expected that these will branch off as the I/O system design matures.

8. Programming Languages and Tools

The PL group is responsible for researching, designing and implementing new language features and associated tools that reflect and support Singularity's overall design goals. The primary language features to be supplied are processes, channels, threads, and a module system tying everything together. In order to enable tools and humans to reason about programs, the programming language or environment supports various forms of partial specifications for all of these features. The implementation vehicle for the programming language work is the Spec# compiler. An important goal of the language work is to leverage the semantic and architectural guarantees to provide light-weight and efficient constructs without sacrificing reliability. For example, communication between processes should support data hand-off.

A good programming language empowers programmers to write efficient, verifiable, and maintainable code. Specifications in various forms enable optimizations, modular verification, and independent evolution of code. The language subgroup is responsible for researching, designing and implementing new language features and associated tools that reflect and support Singularity's overall design goals. The language features are to be designed to enable static verification as far as possible, while leveraging the semantic and architectural guarantees to provide light-weight and efficient implementations without sacrificing reliability.

Channels are the fundamental interaction features in Singularity. Processes communicate only via channels. Channels manifest themselves as channel endpoints within a program. A program interacts with an endpoint similar to the way a program interacts with an API. It is fair to say that in Singularity, cross process APIs take the form of channels, and the contract on the channel describes the valid interactions with the API. Due to their central position, channels must support efficient transfer of data between processes

Within a process, we envision multiple threads of control. Such threads are interesting to allow light-weight concurrency as well as sharing data in memory. However, in order to reason about code's adherence to contracts, it will be necessary to also control how threads share data and synchronize. The thread features in the programming language thus also provide support for specifying what is not shared, what is shared and how various threads interact with the shared data.

Modularity, abstraction, and reuse are central aspects that every programming language should support. We believe there is room for improvement with regard to these in C#. The PL group will thus design architectural and other mechanisms in the form of a module system to support separation of concerns and modular verification.

The implementation vehicle for the programming language work is the Spec# compiler. Verification tools may be either integrated in the compiler, or separate. In general, we don't view any particular compiler as being required to build Singularity code. In fact it is necessary that all contractual obligations, from memory safety, over channel contracts, to thread contracts, be verifiable at the IL level. Thus, it is part of the groups charter to define the associated contractual metadata, and how the verification is performed at the IL level. This allows code to be written in any language, including directly in IL, as long as it comes with the necessary metadata and can be verified.

9. Metadata

Singularity stores and uses metadata deeply and pervasively. These metadata express a variety of type information, configuration information, administrative information, and system information. The Metadata group will consider the initial content, structure, and interrelation of Singularity's metadata.

We will then design and implement an extensible metadata system with unified mechanisms for creating, storing, retrieving, sharing, and manipulating metadata in a consistent manner. Our goal is a simple, efficient design and implementation that can also serve as a general-purpose building block for other parts of Singularity.

We can think of Singularity as a strongly-typed environment, in which applications can communicate only over channels with enforced types and enforced contracts. Local configuration settings list the installed applications, and local administrative rules establish a security policy. These types, contracts, lists, and rules are among the many forms of metadata in Singularity.

The charter of the Singularity Metadata group is to consider the design of an extensible metadata system with unified mechanisms and a consistent interface for structuring, creating, storing, retrieving, checking, and sharing metadata in Singularity.

Since the Singularity design is still beginning, we cannot yet name all the kinds of metadata that Singularity will store and use. Moreover, since Singularity is a research environment that we hope can evolve in unexpected directions, there may never be a final list. For now, we will start by naming some initial kinds of metadata we know we will need in Singularity, and use this list to explore ways to structure a metadata store with mechanisms for creating, storing, retrieving, checking, and sharing metadata. We can also consider ways to generalize our idea of metadata as a building block that other services and applications in Singularity can use.

10. Monitoring and Diagnostics

Goal: Design and realize infrastructure for making singularity applications testable for functional and non-functional properties. Provide means for adaptive execution monitoring and execution control. Also, provide methodology and possibly language/compiler support for writing testable applications. All efforts are focused on managed code execution only.

Features captured: Debugging, Profiling (of various parameters), Runtime-Verification, and Program Exploration

Techniques: Dynamic (adaptive) code instrumentation, state capturing, backtracking, …

11. Power Management

The group is responsible for measurement and management of power consumption by different hardware components. Specifically, to track device state changes and device activities that are relevant to power consumption and to alter device state where possible to meet power management goals. These goals depend on the hardware platform, but broadly they are: extending battery lifetime, reducing noise, keeping devices below critical thermal thresholds, reducing electricity costs, and reducing cooling costs. The initial focus will be on the first two, as most relevant for small devices/home media boxes.

The kind of power-related actions I can imagine are: changing processor clock (dynamic voltage scaling); delaying disk requests to minimize spinups; turning devices and buses off when idle; adjusting fan speeds to minimize noise without overheating. Obviously all of these have performance/functionality implications: we need to craft the power management interface for devices and resource managers so that we have sufficient information to make decisions about voltage scaling, etc., and sufficient control to implement them.

12. Process Architecture

13. Reliability Services

The collection of language constructs, design, programming, development methodologies and constraints, tools, and system features that help prevent program execution from entering an erroneous state, and in the event it does, recovering from, or mitigating the impact of this error on the system

14. Resource Management and Scheduling[2]

The Singularity Resource Management and Scheduler Group's charter is to:

- Make it possible to develop independent real-time applications independently on Singularity, while enabling their predictable concurrent execution, both with each other and with non-real-time applications.
- Enable practical management of multiple resources by applications, including accurate usage accounting and the ability to reserve resources.
- Support scheduling and resource management for non-real-time interactive applications as well.

Note that scheduling and resource management are cross-cutting issues, affecting many aspects of the system, such as memory management, GC, I/O, etc. Initial resources we are considering support for are CPU, I/O bandwidth, network bandwidth, and power/energy. It is explicitly outside the scope of the group to support safety-critical hard-real-time applications.

15. Security

The Singularity security group is responsible for the following aspects of the Singularity design:

- an identity model that specifies how principals are assigned to processes;
- a policy model that determines how an authorization policy can be defined, set and changed for an object (where "object" includes both channel endpoints and anything that can communicate through one);
- an authorization model that specifies how to answer the question, "does a process with principal x have access to object y?"

The group's design goals include:

- leveraging the restrictiveness and checkability of Singularity's design to generate simple, understandable and (where feasible) verifiable identity, policy and access models;

[2] The Power Management group is currently working as a subgroup of the Resource Management and Scheduling group.

- simplifying policy specification in particular by better understanding the space of policies used or preferred in practice (again, in the spirit of the "Singularity razor"); and

- incorporating up-to-date understanding of security technology (such as cryptography and sandboxing) and its applications (such as multi-domain distributed systems security and partially trusted code) "from the ground up".

The group is also responsible for implementing mechanisms that allow Singularity to conform to the above models. These mechanisms may include:

- management of identities, both within a machine and across machines--their creation, recognition (authentication), and association with processes;

- authorization policies that can be set and changed both declaratively (through an administrative interface) and programmatically (via an API);

- authorization verification functionality for basic system services and resources; and

- support for confidentiality of communication across channels that might be prone to eavesdropping (such as between machines).

More specific, nearer-term goals include:

- a proposed definition of principals (including an API for creating and managing them);

- a proposal for mapping principals to processes and a set of default rules for identity inheritance in process creation (and an associated API for setting the principal for processes);

- a proposal for user session establishment; and

- an authorization API suitable for process management, IPC/channel security, and Metadata and file system access control.

16. Virtual PC

The Virtual PC Group is responsible for providing support for legacy code running in a Virtual Machine hosted by Singularity. The group will port the VPC world-swap mechanisms into the Singularity kernel, and the virtualized VPC hardware into a Singularity process. The group will attempt to do so in a manner that allows Singularity with a Virtual PC to run as a guest system within another Virtual PC to facilitate the development of Singularity. The VPC Singularity process will run as non-safe binary code since we are not going to port to Spec# (in the near future).

Later, the Virtual PC Group will investigate:

- Running legacy device drivers and OS components in VPC.

- Running multiple instances of VPC machines on Singularity.

- Ensuring VPC doesn't adversely affect Singularity's security and stability.

- Para-virtualization for performance.
- Model-checking and other analysis that leverages VPC.

17. User Applications

The User Application Group is responsible for finding real applications to run on Singularity. The primary objective of running real applications is to keep the design and implementation of Singularity "honest", by making us take into account real applications. The second objective of running real applications is to help guide and prioritize implementation decisions. While VPC support in Singularity will hopefully allow running existing applications, general application compatibility is a non-goal for the Singularity API.

Applications under consideration include:

- Personal Video Recorder,
- Home firewall,
- Audio PC,
- Home storage server,
- Telephony and VOIP,
- Video conferencing,
- Car synchronization.

