Singularity Design Note

0

Design Motivation

A New Platform for OS Research and Reliable Systems

Singularity is a managed code operation system kernel built in Microsoft Research. This design note describes the motivation and priorities for building Singularity. Other design notes describe the abstractions and implementations of Singularity features.

1. Motivation

The Singularity project attempts to revive OS research both at Microsoft and in the larger OS community.

The Singularity research team defines operating system research as research into the base abstractions for computing and research into implementations of those abstractions as exposed by the OS. By returning to this basic definition of OS research, Singularity embraces the opportunity to re-think OS abstractions and their implementations.

OS research is ready for a revolution. Modern systems are bound by abstractions defined in the early 1970s. OS research has not kept pace with changes in application composition or security needs of everyday usage scenarios. To a lesser degree, OS research has not exploited the exponential evolution of hardware exemplified by Moore's Law. For example, operating systems are largely ignorant of the proliferation of processing cores beyond the CPU.

2. Priorities

2.1. A Research OS

Singularity is first and foremost a research system. Singularity exists to enable prototyping and evaluation of new OS, language, and tools research. The primary benefits of Singularity for OS, language, and tools research are minimalism, design clarity, and extensive utilization of modern languages and tools.

Singularity does not try to implement all of the features of either Windows or other commercial grade operating systems. Singularity tries to implement features sufficient to demonstrate the viability of research innovations. To create an unencumbered research platform, Singularity eschews compatibility.

The Singularity architecture is small enough to be readily understood in its entirety and to allow individual researchers to conceive and implement new OS, language, or tool innovations on the code base in a reasonable period.

In order to maximize use as a research platform, Singularity favors design clarity over performance or compatibility. At most decision points Singularity attempts to provide "good enough" performance, but no better. By default performance is secondary to other research targets such as security, reliability, or soundness of design; exceptions to this rule occur only where performance is the primary focus of specific research.

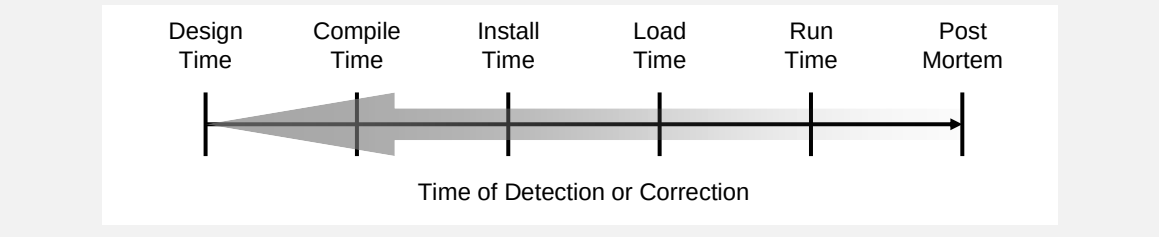
2.2. Building a Reliable System

The greatest opportunity for OS advancement is in system reliability. System reliability can be expressed loosely in terms of a system that "just works", or more concretely as a system that never behaves in a manner unexpected or unanticipated by its designers, developers, administrators, or users.

The lack of reliability in current systems is most apparent in their susceptibility to unanticipated security vulnerabilities, their instability due to third party code, and their uncertain behavior in the face of software installation, update, or removal.

System reliability is not the same as system availability. A reliable system might be offline for extended periods (say 15 minutes a week, a system availability of .999) provided the unavailability occurs at an expected time under expected conditions. With a reliable system, users will know well in advance when it will be offline, how long it will be offline, why it is going offline, and that when the system comes back online that it will either be in the new target state or in the old state with a clear understanding of why it could not make the transition.

Singularity improves system reliability by attempting to mitigate sources of unreliability as early as possible in the life of a system. Ideally, many sources of system unreliability will be removed at design time or compile time through aggressive application of process, tools and language features. Those sources of unreliability that cannot be removed at design or compile time should be removed at install time or load time. Only in the worst case should sources of unreliability be removed at run time. Under no circumstances should sources of unreliability go unhandled, resulting in unexpected user visible failure which can only be analyzed after the fact.



The primary goal of Singularity research is move as many sources of unreliability and their mitigation as far to the left as possible on the access time. For example, moving the detection and correction of incorrect DLL binding from load time and run time to install time would eliminate "DLL Hell".

System reliability is strongly determined by the ability of both OS and application programmers to produce reliable code. Singularity relies heavily on tool innovation and design simplicity to produce a system in which it is hard to create unreliable programs.

3. Centers of Gravity

Four design points combine to produce an OS that is agile to future research and innovates in system reliability. These design points are: a type safe **abstract instruction set** as the system binary interface, a **unified extension mechanism** for applications and the OS, a **strong process isolation architecture**, and a **ubiquitous metadata infrastructure** describing code and data.

Based in Safe MSIL, a type safe abstraction instruction set provides an ideal system binary interface. It eliminates a whole class of programmer errors due to bad pointer arithmetic, enables changing boundaries between privileged and unprivileged code, opens new opportunities for dynamically adjusting trade-offs between security and performance, and allows ubiquitous analysis and instrumentation for both reliability and research. Incorporating the abstraction instruction set into the OS also enables the elimination of the current line between OS and VM runtime.

Providing a unified extension mechanism simplifies design and implementation of the OS, applications, and new research proposals. Supporting one extension mechanism, instead of many, allows focus to get that one mechanism correct in design; it also simplifies the management of extensions from the perspectives of both reliability and security. Choosing an extension mechanism that isolates extension code simplifies the development of reliable systems and opens otherwise unavailable opportunities for static analysis and optimization.

Strong process isolation overcomes two major challenges in contemporary systems: unintended collision between friendly applications and undesired attack from hostile code. The need for strong process isolation is witnessed by the rise of virtual machines like Virtual PC and OS-alternative security models such as the CLR's Code Access Security[1]. Strong process isolation will improve system reliability and provide solid boundaries which can be exploited for further OS research.

Ubiquitous metadata provides a unifying design thread and enables the removal of unreliability as early as possible in the system life cycle. Metadata enables type safety and flexibility in the abstraction instruction set, verification of interfaces in the extension model, and explicit relationship labeling for deep process isolation. Introduction of new classes of metadata via an application abstraction enables moving checks for unreliability from run time to installation or even compile time. Singularity enables OS research across the application boundary by providing a ubiquitous, OS-protected infrastructure for storing and manipulating metadata.

Initial research and development resource will be devoted to making progress on these four centers of gravity as early as possible in the design and implementation of Singularity.

[1] This use of Code Access Security (CAS) references the implementation of stack walking, etc., in the CLR, not the more generic concept of attaching a security identity to a specific piece of code based on either its content or a code signer. The more generic concept is desirable and possibly a required feature in Singularity.

