Broad New OS Research: Challenges and Opportunities

Galen C. Hunt[1], James R. Larus[1], David Tarditi[1], and Ted Wobber[2]
[1]*Microsoft Research Redmond, Redmond, WA 98052, USA*
[2]*Microsoft Research Silicon Valley, Mountain View, CA 94043, USA*
http://research.microsoft.com/os/singularity

Abstract

Contemporary software systems are beset by problems that create challenges and opportunities for broad new OS research. To illustrate, we describe five areas where broad OS research could significantly improve the current user experience. These areas are dependability, security, system configuration, system extension, and multi-processor programming. In each area we explore how contemporary systems fall short. Where we have thought of possible solutions, we offer directions for future research.

Finally, we describe Singularity, a research project at Microsoft Research that is building a new operating system to explore four of these challenges. Singularity incorporates three specific design decisions in order to increase system dependability and improve system security, configuration, and extension. These design decisions include the adoption of an abstract instruction set as part of the system binary interface, a unified extension architecture for both the OS and applications, and a first-class application abstraction.

1. Introduction

The products of forty years of OS research are sitting in everyone's desktop computer, cell phone, car, etc.—and it is not a pretty picture. Modern software systems are—broadly speaking—complex, insecure, unpredictable, prone to failure, hard to use, and difficult to maintain. Part of the difficult is that good software is hard to write, but in the past decade, this problem and more specific shortcomings in systems have been greatly exacerbated by increased networking and embedded systems, which placed new demands that existing architectures struggled to meet. These problems will not have simple solutions, but the changes must be pervasive, starting at the bottom of the software stack, in the operating system.

Unfortunately, as the emergence of the Internet exacerbated problems in conventional systems, the research community turned its attention from broad OS research to focus on incremental improvements or new areas such as distributed systems [17].

Without OS solutions, others stepped into the void by devising partial, application-level solutions to these problems. Consider, for example, the problem of isolating code for potentially untrusted sources. Applications and programming language runtimes have tried to supplant inadequate OS security with partially redundant and complex security abstractions using stack walking and code signing [12][24]. Others have attempted to solve this problem by go so far as to replicate entire operating systems in virtual machine monitors [11]. While the engineering is admirable, one wonders if the OS could provide a more integrated solution.

The remainder of this paper has three parts. Section 2 suggests example areas in which OS research could make operating systems work significantly better for most users. We offer these areas as evidence of opportunity, not as an exhaustive research agenda. Section 3 describes work in the Singularity project to address some of these areas. Finally, Section 4 summarizes the challenges and opportunities for broad new OS research and draw conclusions.

2. Opportunities for OS Research

To suggest the many opportunities for OS research, we list five areas in need of new ideas and abstractions: dependability, security, system configuration, system extension, and multiple processor programming. This list is intended to be illustrative, not exhaustive.

2.1 Dependability

A system is *dependable* if it behaves predictably and reliably; in other words, if its behavior consistently conforms to an understandable and useful model. A system's *perceived dependability* is a function of both user expectation and actual system behavior.

Unfortunately, the perceived dependability of contemporary software systems is low, particularly in the eyes of non-technical users [15].[i] Partially this results from raw software failures. However, it also results from unpredictable system behavior.

Broadly speaking, the owner of a modern PC encounters frequent unexpected behaviors. By contrast, most modern cars are considered quite dependable by their users; this despite the fact that cars can require as much as one hour of maintenance for every one hundred hours of usage.[ii] We claim that modern cars are considered dependable because they have an easily understood operation model consisting of regular fueling, regular oil changes, regular maintenance, and basically predictable, uninterrupted usage the rest of the time.

No open, general purpose software system can make a similar claim. They all must be patched frequently and regularly to fix flaws that open the system to malicious attack. They all can fail in ways that are inexplicable and unpredictable to ordinary users. Many of these users are afraid to change their system in even the slightest way, for fear of breaking them.

2.2 Security

Contemporary OS security systems were designed to protect users of a system against each other and to protect the OS from errant programs. These security architectures were developed in the quaint past when code came from trusted sources and networks connected us with our friends and colleagues. In today's connected world, users and computers are surrounded by unscrupulous advertisers, petty criminals, and increasingly organized crime. In this world in which executable code can and does come from anywhere, the OS needs to protect user and system resources from potentially hostile code that a user runs either intentionally or unintentionally. This is a very hard problem given that desired code may do useful work!

To bring code into an OS security model, there must be a basic OS abstraction that represents the identity of code. The abstraction should also capture the provenance of the code as well as provide a means for checking code integrity. Once code is identifiable, we can imagine enforcing security policy pertaining to it.

Code identity alone, however, is not sufficient. Software components interact in exceedingly complex ways, and many such interactions are security-relevant. We can expect the next generation of attacks to exploit unplanned and unprotected interactions between software components. There is fertile ground for research in understanding how to prevent such attacks by design.

The Java [12] and Common Language Infrastructure (CLI)[iii] [24] programming environments have explored some of these issues. However, the security models in these systems are complex and largely separate from OS models.

2.3 System Configuration

Contemporary operating systems contain abstractions for many components of modern applications, such as processes, threads, and shared libraries, but applications and their dependencies are only informally characterized. Lacking a strong concept of an application's complete configuration, the OS has no mechanisms to guarantee the integrity or provenance of an application. A system is only as stable as its most fragile component, which cannot be identified in current systems; systems which provide no easy way to distinguish application components intermixed in file systems and configuration registries.

Consider, for example, the case of applications colliding in their usage of shared spaces such as file systems or configuration registries. The installation of one application may corrupt or irreversibly alter the configuration of another via changes to a file or registry. The "DLL Hell" problem in Windows systems occurs when one application overwrites a common shared library with a version incompatible with an existing application. Similar problems can occur when an application overwrites configuration information mapping from document extensions to applications. To compensate for the absence of OS managed applications, users resort to ad-hoc application isolation techniques, such as jails [14] or virtual machine monitors, such as VMware [9] and Xen [3].

2.4 System Extension

Since no monolithic system can satisfy all users, most complex software lets users load code to extend functionality. Dynamically loaded extensions are found as widely as device drivers in kernels and spelling checkers in word processors. Whether in the OS or an application, most extensions are loaded directly into a host address space with no hard interface, protection boundary, or clear distinction between host and extension code. Extension through in-process code loading appears flexible and attractive, but due to a lack of isolation, extensions are a major source of software reliability and security problems. For example, faulty device drivers cause a large fraction of Windows and Linux failures [22].

A number of OS research efforts, including Exokernel [13], SPIN [5], VINO [21], and Nooks [22] have sought safer OS extension without addressing the more general problem of application extension. Pragmatically, each of these systems provided domain-specific models for OS extensions. Software fault isolation (SFI) [23], one of the few research efforts to consider application extension, limits an extension to a subset of an application's address space. However, the overhead for SFI is quite high and still exposes published data structures to corruption by the extension.

In Section 3.1.2, we will describe research in the Singularity system to create a unified extension architecture for both the operation system and applications.

2.5 Multi-processor Programming

Thanks to the physical constraints of semiconductor device scaling, it has become easier to replicate processors than to increase processor speed. Over the next decade the number of processing cores per chip could double every 18-24 months. Processing cores are replicating not only on CPUs, but in peripheral devices as well. Notwithstanding recent work on scheduling algorithms for multi-core CPUs [10] and programming

GPUs [6], there are research opportunities to create new abstractions for programming large numbers of processors and to treat the non-CPU processors found in graphics, network, and storage devices as first-class compute resources.

3. Singularity

Singularity is a Microsoft Research project to develop techniques and tools for building dependable systems that address the challenges faced by contemporary software systems. Singularity is approaching these challenges by simultaneously pushing the state of the art in operating systems, run-time systems, programming languages, and programming tools—the foundation on which software is built. The Singularity OS is first and foremost a research system. Singularity strives for minimalism and design clarity, and makes extensive use of modern languages and tools.

By plan, performance is secondary to other research objectives such as security, dependability, and soundness of design. However, in places where we believe performance is central to the research challenge, such as streamlining cross-process communication, we strive for high performance solutions that also meet the other objectives.

To increase our ability to conduct a broad new OS research agenda, we have forgone compatibility with previous operating systems. Our experience is that new abstractions are best developed in an environment free of contradictory legacy requirements and then ported to legacy environments when the abstractions have matured. We recognize that this is a calculated risk; in the longer term, we have made provisions to implement a virtual machine monitor in Singularity as legacy support becomes a requirement.

3.1 Design Choices

A key focus of Singularity research is improving system dependability. Singularity improves dependability by dramatically increasing the scope of sound verification techniques to detect sources of unexpected system behavior. To broaden the scope of sound verification techniques, Singularity fixes the behavior of system components as early as possible in lifetime of their code (see Figure 1). To lengthen the scope of sound verification techniques, Singularity constrains system organization and preserves metadata so that verification results can be applied even to late-bound composites.

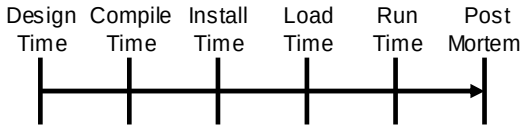

Figure 1. Code lifetime of a software component.

Singularity incorporates three key design choices to improve system dependability. These design choices are: an abstract instruction set as part of the system's application binary interface (ABI), a unified extension architecture, and a first-class application abstraction. The abstract instruction set provides the OS with a flexible layer of indirection between application code and a processor's instruction stream. The unified extension architecture enables rich, inexpensive, *and* safe interaction between system components. The application abstraction enables OS management of applications and integration of applications into the security model as security principles.

Early indications are that these design choices also have a positive impact on the challenges of system security, configuration, and extension. System security and configuration in Singularity are given much deeper treatment by Abadi *et al.* [1] and DeTreville [8], respectively.

3.1.1 Abstract Instruction Set

Singularity executables represent executable code in an abstract instruction set, called MSIL. MSIL is Microsoft's implementation of the ECMA Common Intermediate Language [25]. All third-party executables, including applications and device drivers, are delivered to Singularity as type-safe MSIL binaries.

Singularity requires that all user MSIL be type safe, which eliminates an entire class of programmer errors due to erroneous or malicious pointer arithmetic. Because Singularity controls the translation of MSIL into processor instructions, the OS retains the opportunity to insert trusted instruction sequences into the unprivileged, but verified, instruction stream. The abstract instruction set also opens new opportunities to dynamically adjust the trade-offs between security and performance, and it allows rigorous analysis and instrumentation of application code.

3.1.2 Unified Extension Architecture

Singularity provides one extension architecture for the operating system *and* applications. Like previous micro-kernels [2][16][18], Singularity incorporates a process-based extension model. Singularity, however, assumes a more aggressive closed-process architecture for both OS and application extensions.

Singularity processes are closed worlds in two regards. First, Singularity disallows shared memory between processes; Singularity processes exchange data exclusively through messages, which are visible to only one process at a time. Second, once execution begins within a process, no new code may be added to the process. Singularity disallows both loading of new code modules and generation of new code into an existing process.

Any OS or application extension code can be loaded only into a child process, separated by a strong isolation boundary. Communication between host and extension across the process isolation boundary is restricted to verified message-passing *channels*. Channels are strongly typed with *contracts*. All cross-channel interactions and contracts are statically verified using a technique called *conformance checking* [7]. Conformance checking guarantees that a contract is fully specified, that two parties communicating through a contract will not deadlock, and that neither party will receive an unexpected message.

By disallowing dynamic loading of new code into a process, Singularity processes become a closed world in which analysis tools can make sound assumptions about process states, invariants, and valid state transitions. The closed-world extension architecture opens new opportunities for static analysis and optimization.

3.1.3 Application Abstraction

Singularity raises the notion of an application to a first-class OS abstraction. Applications have security identities and signed manifests declaring their constituent components. Installation, maintenance, and removal of applications are all operations controlled by the OS.

Applications are strongly isolated. Access to shared resources—including other applications—is mediated through the Singularity security model. The security model uses code identity and component relationships in access control checks [1].

The application abstraction is recursively applied to the OS itself, with the kernel and other OS components described by manifests. Manifests form the roots of a metadata infrastructure that enables introspection across the entire system –both applications and operating system. Through this metadata it should be possible, for example, to examine an offline Singularity system image and determine if it has the necessary components and configuration to run on a specific hardware configuration or host a specific application. A specific Singularity system as represented by an installation image then becomes a *self-describing artifact*, not just a collection of bits accumulated with at best an anecdotal history.

Most operating systems install and uninstall applications through imperative updates to mutable configuration information held in the file system and in configuration registries. We expect to extend Singularity's application abstraction to support a declarative form of configuration for a whole system, which we expect will eliminate whole classes of system misconfiguration [8].

3.2 Singularity Architecture

Singularity is a type-safe OS. Where traditional operating systems present untyped memory and the hardware instruction set to applications, Singularity replaces these with the abstractions of typed memory and an abstract instruction set, in the form of type-safe MSIL.

Singularity relies on type-safety and control of the translation of the abstract instruction set to machine code to enforce system protection boundaries. This allows faster and more efficient process-to-kernel context switches and communication between processes.

At the heart of the system is a trusted computing base (TCB), see Figure 2. The Singularity TCB is composed of the kernel proper, trusted runtime code, and MSIL translators. The TCB maintains security policies and guarantees that no untrusted or unverified instructions ever execute. The TCB ensures process integrity by providing isolated object spaces for processes and constraining IPC communication to contract-conforming channels.

Most of the TCB is written in Sing#, an extension of C# with specifications on objects and conformance-checked channels. The object specifications come from Spec# [4], which extends C# with pre-conditions and post-conditions on methods, and invariants on class variables. An implementation conforms to a contract if it only sends or receives messages over the channels those message described in the channel and all channel-visible state changes conform to the state machine in the channel contract.

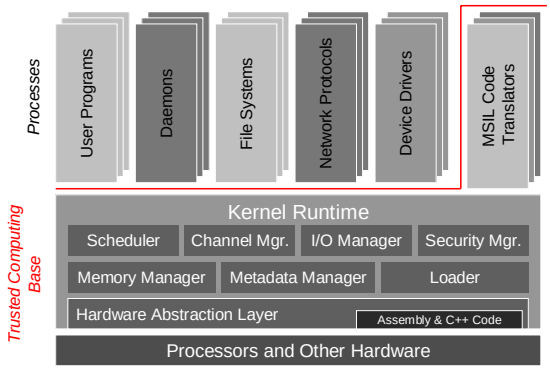

Figure 2. Singularity Architecture.

Portions of the TCB, including the per-process garbage collectors (GCs), are written in unsafe C#. At the bottom of the system, a small body of C++ and assembly code provides the lowest portions of the hardware abstraction layer (HAL). Spec# and C# codes are emitted as MSIL and translated to hardware instructions. The C++ code is compiled directly to the hardware instruction set.

All third-party binaries, including applications, extensions, and device drivers, are delivered to Singularity as type-safe MSIL binaries. Each process receives its own memory pages, but type safety and garbage collection guarantee that no process can hold pointers to any page it does not rightfully own. As a result, most of the sys-

tem, including third-party code, can run in the same address space and hardware protection domain as the kernel.

MSIL binaries may be translated into hardware instruction streams at load or install time based on metadata in the application manifest. Caching of hardware instruction streams is invisible to both applications and users.

The Singularity kernel integrates some of the runtime services of the CLI with traditional kernel-based OS services such as scheduling, IPC, and I/O management. By redrawing the line between the runtime and the kernel, Singularity eliminates redundancies in resource management and security policy. The runtime also enjoys access to kernel features, such as direct access to the processor's MMU.

Singularity's implementation of CLI features is factored to minimize code in the trusted computing base. Code translators reside in processes outside the kernel and convert MSIL into verified hardware instruction streams. The loader caches hardware instruction streams and maps them into processes. The memory manager includes the GC and its accompanying facilities such as the GC write barrier. The metadata manager acts as a repository for traditional CLI code metadata, such as type information required for garbage collection. The metadata manager also coordinates information related to the application abstraction and application manifests.

The Singularity architecture supports multiple MSIL code translators. Individual translators may generate qualitatively different code from the same input. For example, one translator might optimize for performance while another may optimize for security by insert security automata [19] into the code. In the future, additional translators might target secondary processors such as GPUs.

3.3 Project Status

The Singularity system has been under design and development for a little over a year. Although still a work in progress, Singularity is now a recognizable operating system with threads, processes, channels, an I/O subsystem, device drivers, a TCP/IP network stack, a base CLI class library and runtime, and a kernel debugger. Singularity boots on PC hardware using the NVIDIA nForce4 chipset and under the Virtual PC VMM. Notable missing features include a GUI and virtual memory paging. The first version of the application abstraction work is coded, but has not yet been integrated with the rest of the system.

Over the next year, we intend to deploy the Singularity system and a small set of applications into the homes of approximately 50 researchers as a home service appliance. Our test deployment will target non-traditional applications, in particular, applications where the service appliance hosts services provided and managed by multiple third parties. A key objective of the deployment is to measure dependability of the current architecture and to experiment with the application abstraction to automate system configuration.

4. Conclusions

The world needs broad operating system research. Dependability, security, system configuration, system extension, and multi-processor programming illustrate areas were contemporary operating systems have failed to meet the software challenges of the modern computing environment.

The OS research community, in collaboration with researchers from the computer architecture, programming languages, and software tools communities, are well positioned to provide innovative solutions to today's software challenges. If the research community fails to take up this challenge, practitioners will likely provide incomplete solutions developed under competitive duress; the outcome is not likely to be a happy one.

Contemporary operating systems, both proprietary and open source, are constrained by backward compatibility and are unlikely to make the radical changes necessary to improve a typical user's computing experience without clear research guidance. A generation of orthodoxy has led software systems to this unsatisfying state.

We believe the OS research community should embrace this opportunity. We recognize that such opportunity does not come without risk. Many nice research OS abstractions have fallen by the wayside. However, as user dissatisfaction with the status quo continues to rise, unique opportunities may arise for either new operating systems or adoption of new OS abstractions within existing systems.

For our part, the Singularity project is responding to this opportunity by re-examining the fundamental abstractions of software systems through adoption of three design choices: an abstract instruction set, a unified extension architecture, and a first-class application abstraction.

5. Acknowledgements

We thank Martín Abadi, Mark Aiken, Paul Barham, John DeTreville, Orion Hodson, Mike Jones, Nick Murphy, and Ben Zorn for their help preparing this paper. We also thank the anonymous referees for valuable suggestion to improve the content and presentation of this paper.

References

[1] M. Abadi, A. Birrell, and T. Wobber. Access Control in a World of Software Diversity. *Proc. of Hot OS X: The 10th Workshop on Hot Topics in Operating Systems,* June 2005.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for UNIX development. *Summer USENIX Conference*, pp. 93-112, 1986.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. Xen and the Art of Virtualization. *Proc. of the 19th ACM Symposium on Operating Systems Principles*, pp. 164-177, 2003.

[4] M. Barnett, K. R. M. Leino, and W. Schulte, The Spec# Programming System: An Overview. *Proc. of Construction and Analysis of Safe, Secure and Interoperable Smart Devices*, 2004.

[5] B. N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. *Proc. of the 15th Symposium on Operating Systems Principles*, pp. 267-283, 1995.

[6] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brooks for GPUs: stream computing on graphics hardware. *Proc. of the 2004 SIGGRAPH Conference*, pp. 777-786, 2004.

[7] S. Chaki, S. K. Rajamani, and J. Rehof, Types as Models: Model Checking Message-Passing Programs. *Proc. of the 29th ACM Symposium on Principles of Programming Languages*. pp. 45-57, 2002.

[8] J. DeTreville. Making system configuration more declarative. *Proc. of Hot OS X: The 10th Workshop on Hot Topics in Operating Systems*, June 2005.

[9] S. Devine, E. Bugnion, and M. Rosenblum. Virtualization system include a virtual machine monitor for a computer with a segmented architecture. *US Patent*, 6397242, 1998.

[10] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Implementing an OS Scheduler for Multithreaded Chip Multiprocessors. *Work-in-Progress Reports, 6th Symposium on Operating Systems Design and Implementation*, 2004.

[11] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, D. Boneh. Terra: A Virtual-Machine Based Platform for Trusted Computing. *Proc. of the 19th ACM Symposium on Operating Systems Principles*, pp. 193-206. 2003.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java language specification.* Addison-Wesley, 2000.

[13] M. Kaashoek, D.R. Engler, G.R. Ganger, H.M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems, *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pp. 52-65. 1997.

[14] P.H.Kamp and R.N.M. Watson, Jails: Confining the omnipotent root. *SANE 2000.* May 2000.

[15] C. Kaner and D.L. Pels, *Bad Software: What to Do When Software Fails.* John Wiley & Sons, 1998.

[16] J. Liedtke. Toward real µ-kernels. *Communications of the ACM*, 39(9):70-77, September 1996.

[17] R. Pike. Systems Software Research is Irrelevant. *Invited talk*, University of Utah, February 2000.

[18] M. Rozier, A. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. CHORUS distributed operating system. *Computing Systems*, 1(4):305-370, 1988.

[19] Schneider, F.B. Enforceable Security Policies. *ACM Transactions on Information and System Security* (TISSEC), 3 (1). 30-50, 2000.

[20] Secunia, *Statistics of released advisories by project*, http://secunia.com/product, 2005.

[21] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. *Proc. of the 2nd Symposium on Operating Systems Design and Implementation*, pp. 213-228, 1996.

[22] M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems, *ACM Transactions on Computer Systems*, 22(4), Nov. 2004.

[23] R. Wahbe, S. Lucco, T.E. Anderson, and S. Graham. Efficient Software-Based Fault Isolation. *Proc. of the 14th ACM Symposium on Operating Systems Principles*, pp. 203--216 1993.

[24] *Common Language Infrastructure (CLI): Partition I: Architecture*, ISO/IEC 23271:2003.

[25] *Common Language Infrastructure (CLI): Partition II: CIL Instruction Set*, ISO/IEC 23271:2003.

---

[i] Data from security advisories suggest that no contemporary system, either commercial or open source, has a monopoly on dependability problems [20].

[ii] An oil change (1 hour) every 5,000 miles (100 hours at 50 miles/hour) is typical and does not take into account other preventive maintenance, which typically takes a car out of commission for an entire day.

[iii] Microsoft's commercial implementation of the CLI is known as the Common Language Runtime (CLR). The CLR is the core of Microsoft's .NET Framework.