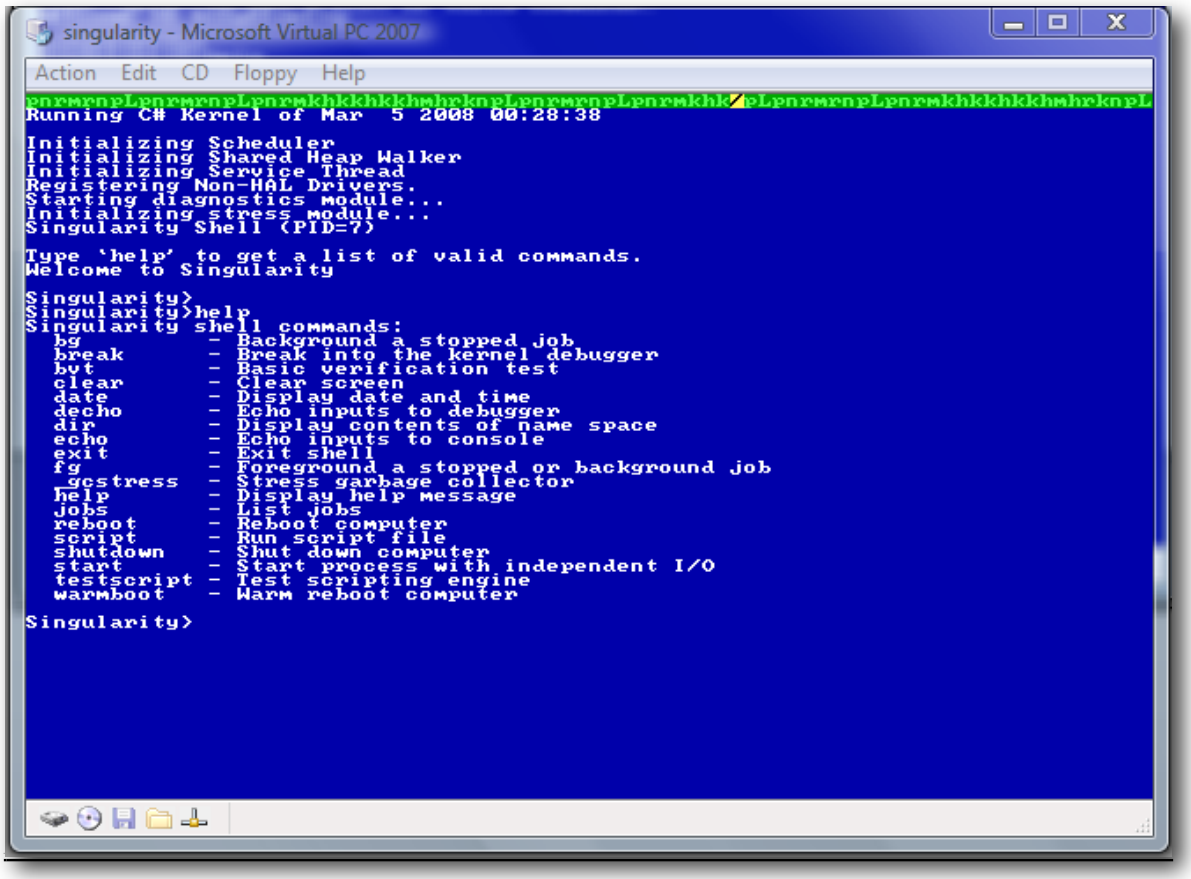
Technische Universität Ilmenau
Fakultät für Informatik und Automatisierungstechnik
Fachgebiet für Verteilte Systeme und Betriebssysteme
Hauptseminararbeit im Themenkomplex „Sichere Betriebssysteme“ im Wintersemester 2008/2009 (Abgabe: 02.02.2009)
Sebastian Wieseler sebastian.wieseler@tu-ilmenau.de

# Singularity unter Betrachtung der Sicherheitseigenschaften




## Abstract

Als *Singularität* wird im Allgemeinen eine Einzigartigkeit bezeichnet. In der Physik, im Speziellen, bezeichnet es Gegebenheiten unter denen physikalische Gesetze nicht definiert sind - wie zum Beispiel in *schwarzen Löchern*.
Im Folgenden behandelt dieses Paper ein Forschungsbetriebssystem von Microsoft - **Singularity**. Es wirft ein Blick darauf, wie die Entwickler versuchen scheinbar alltägliche Gesetze von Instabilitäten und Unsicherheiten in alt bekannten Betriebssystemen außer Kraft zu setzen.
Sie beschreiten den „alten Weg“ des *Managed Codes*, den man beispielsweise aus der Java-Welt bereits kennt. Programme werden dabei in einer virtuelle Umgebung gelagert, welche einen umfassenden Isolationsschutz bieten soll. Mit Microsoft Virtual PC 2007 funktionieren als Host-Betriebssystem beispielsweise Windows XP oder Vista.

[1] Bildquelle: Jon Davis. Singularity Now Open To The Public. http://www.jondavis.net/techblog/post/2008/03/05/Singularity-Now-Open-To-The-Public.aspx. 2008

## 1. Kurzer Überblick über Singularity

Singularity ist ein Microsoft Research Projekt. Es wird aktiv als Forschungsbetriebssystem entwickelt seit etwa 2004. Seit dem März 2008 ist es quelloffen. Man kann es jedoch nicht als Open Source bezeichnen, da es nicht im kommerziellen Bereich weiter entwickelt werden darf.

Microsoft setzte sich als Ziel Robustheit, Stabilität, Zuverlässigkeit und Sicherheit in seinem neuen Betriebssystem zu verwirklichen. Es sollte außerdem für den praktischen Einsatz geeignet sein und weiterhin Systemressourcen schützen unter der ausschließlichen Zuhilfenahme von Softwaremechanismen.

Aktuell umfasst es 300.000 Zeilen Quellcode, welche Netzwerkkarten-, IDE-, Sound-, Tastaturtreiber und einen TCP/IP-Netzwerkstack beinhalten. [WP05+]

## 2. Einordnung von Singularity in bekannte Betriebssystemkonzepte

Für die Einordnung in die Mikrokernel-Welt würde dafür sprechen, dass der Kernel Basisfunktionalitäten wie das Page-, I/O-, Channelmanagment und das Scheduling bereitstellt. Außerdem liegen andere Systemkomponenten außerhalb des Kernels. Prozesse und Betriebssystemkern teilen sich insbesondere keinen Speicher. Beispielsweise gilt dies besonders für (Geräte-)Treiber. [HAF+07]

Singularity könnte aber auch als monolithisches Betriebssystem eingeordnet werden, da alle Prozesse im „Kernel Mode" (Ring Null) laufen. Dadurch entfallen z.B. teure Kontextswitche, sowie Kopieroperationen für diese.
Microsofts neues Betriebssystem setzt die erforderliche Prozessisolation ausschließlich in Software, in so genannten *SIPs*[2], durch. Dadurch sind prinzipiell keine Hardwareunterstützungen wie CPU-Ringe oder *MMU*[3] mehr erforderlich für die Durchsetzung dieser Isolation. [Ess06]

## 3. Wiederholung von Grundkonzepten

Als *Typsicherheit* bezeichnet man die Vermeidung von Typverletzungen. Darunter fallen unzulässige Casts. Das bedeutet, dass Castoperationen zu nicht verwandten Typen verboten sind. Desweiteren muss sichergestellt werden, dass ein unsauberes (Funktions-)Überladen verboten ist, sowie dass Typen immer korrekt interpretiert und manipuliert werden.
Eng verknüpft ist damit auch die *Speichersicherheit*. Sie unterbindet den willkürlichen Zugriff auf Speicheradressen. Referenzen sind auch nur auf gültige Datenobjekte erlaubt. Weiterhin ist eine Speicherverwaltung mittels Zeiger untersagt. Die Dealokation wird nicht wie gewohnt von Hand, sondern via *Garbage Kollektoren* erledigt werden. [LHT]

---

[2] *softwareisolierten Prozessen*

[3] *Memory Management Unit*

## 4. Erster Einblick in die Konzepte von Singularity

Singularity wurde in einem C#-Derivat geschrieben, was von den Singularity-Entwicklern für dieses Projekt entworfen worden ist: *Sing#*. Diese Sprache soll Typ- und Speichersicherheit garantieren. Außerdem bietet diese Sprache eine Unterstützung für die nachrichten-orientierte Kommunikation. [FAH+06]
Es wird sichergestellt, dass die Ausführung nur erfolgt, wenn der Code typ- und speichersicher ist. Um dies zu garantieren finden die Überprüfungen zur Kompilierung statt.
Die ausführbaren Dateien liegen dabei als *MSIL*[4] Binaries vor. Diese Zwischensprache ist vergleichbar mit dem Byte Code von Java. Das Vorhandensein ist besonders wichtig, da man bei 3rd-party Anwendungen nicht davon ausgehen kann, dass diese im Quellcode vorliegen. Dabei wird durch die MSIL auch die Typ- und Speichersicherheit überprüft werden können. [HL04]

Von einer *Sealed Process Architecture* [HAF+07], wie sie bei Singularity vorliegt, spricht man, wenn vier Invarianzen gelten.
Zum einen die *Fixed Code Invariant*, welche sicherstellt, dass keine Veränderung des Codes zur Laufzeit stattfindet. Etwaige Garantien kann beispielsweise dafür das Betriebssystem geben. Zum zweiten existiert die *State Isolation Invariant*, welche eine strickte Zustandsisolation durchsetzt. Dies bedeutet, dass ein Zustand von einem Prozess nicht von einem anderen direkt modifiziert werden kann. Es herrscht damit vollständige Isolation zwischen den Prozessen. Eine implizite Kennung von Absender und Empfänger fordert die *Explicit Communication Invariant.* Dabei findet alle Kommunikation über explizite Mechanismen statt. Die *Closed API Invariant* stellt sicher, dass das Betriebssystem keine API-Funktionalität bereitstellt (für unprivilegierten Prozesse), die gegen andere Invarianzen verstoßen könnte. Beispielsweise darf ein Prozess keine Assembler-Befehle ausnutzen um gegen die Speichersicherheit zu verstoßen oder über andere „Kanäle" mit Prozessen zu kommunizieren.

## 5. Schützenhilfe vom Compiler

Die Entwickler von Singularity haben sich das Ziel gesetzt die Isolation und das Durchsetzen von abgeschlossenen Coderäumen ausschließlich in Software umzusetzen. Dabei soll die Speicher- und Typsicherheit garantiert werden.
Das Compiler-Frontend übersetzt dabei den Quellcode zur MSIL. Dabei finden zahlreiche Überprüfungen statt, die sicherstellen sollen, dass das MSIL-Binary anschließend die gewünschten Eigenschaften besitzt. Danach wird der Sing# Compiler *Bartok* die MSIL zu x86-Code kompilieren. Dabei können verschiedene Optimierungen stattfinden. So ist es Bartok beispielsweise möglich durch *tree shaking* nicht gebrauchte Funktionalitäten wie z.B. unbenutzte Klassen, Methoden und sogar Felder zu eliminieren. [HLA+05]

Desweiteren erreicht man durch die abgeschottete Welt mehr Analysemöglichkeiten von Prozesszuständen und Zustandsübergängen, wie in [HLT+05] beschrieben.
Es gibt zahlreiche Konsequenzen, die daraus folgen, dass Typ- und Speichersicherheit zur Kompilierung durchgesetzt werden. Es ist dadurch kein Nachladen von z.B. Bibliotheken und Klassen zur Laufzeit möglich. Plugins können daher nicht auf normalen Weg nach Singularity portiert werden. Sie könnten, genauso wie Gerätetreiber, in inkonsistenten oder korrupten Zuständen enden oder durch Ausfälle Speicherbereiche unaufgeräumt zurücklassen.

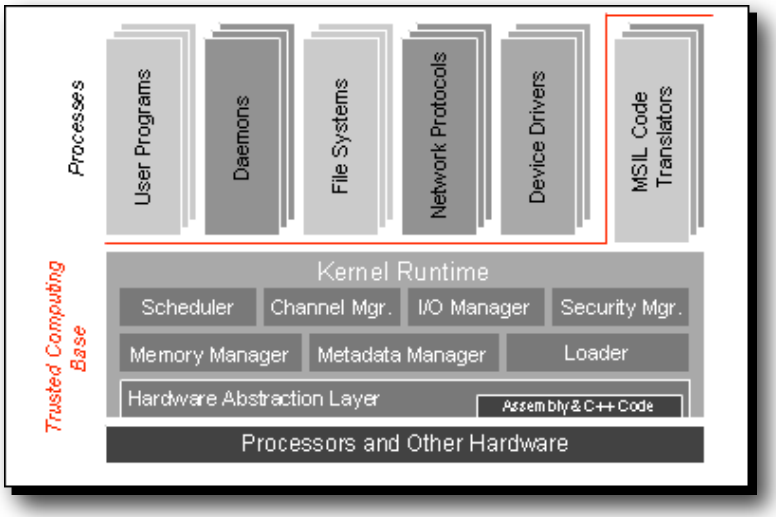
---

[4] *Microsoft Intermediate Language*

Gedanken sollte man sich auch zum bisherigen Konzept des *Shared Memorys* machen. Eine Kommunikation mittels dieser Technik ist in Singularity ebenfalls nicht möglich. Durch die Durchsetzung der strickten Isolation muss man sich anderweitigen Methoden bedienen.
Durch die SIPs erreicht man einen Zustand, in dem man alles im Ring 0 auf der CPU laufen lassen kann. Das heisst jede Ausführung läuft praktisch im „Kernel Modus". Man spart teure Kontextswitche und das Flushen der *TLB*[5] ein, sowie „unnötige" Cache-Misses. [HL07]

### 6. MSIL als sicherer Boden?

[HLT+05] beschreibt die *TCB*[6] als Herz eines Systems. Sie umfasst in Singularity neben dem Kernel und dem vertrauenswürdigen Laufezeitcode zusätzlich den MSIL Übersetzer und Verifizierer. Sie setzt Sicherheitspolitiken durch und gibt Garantien für „sicheren" Code. Dabei führte man die MSIL aus Zwecken der höheren Abstraktion ein. Sie beinhaltet zum Beispiel Typen und Objekte. Diese sind (leichter als x86 Code) verifizierbar auf Typ- und Speichersicherheit. Deshalb ist der MSIL-Übersetzer Bartok Teil der TCB, wobei beispielsweise der Sing#-Compiler nicht notwendigerweise Teil sein muss. [HAF+07] folgert daraus eine Unterscheidung in verifizierten und vertrauenswürdigen Code. Ersteres erreicht man durch die MSIL, zweiteres erstreckt sich ausschließlich auf *HAL*[7], den Kernel und Teile vom Laufzeitsystem. Dies folgt daraus, dass einige Teile dieser Komponenten in Assembler, C++ oder „unsicherem" C# geschrieben sind und daher vom System als vertrauensvoll ohne Überprüfung eingestuft werden müssen.



[HLT+05]: Singularity Architecture

Als zukünftig verbesserungswürdig gibt [HAF+07] den Umstand an, dass man der korrekten Übersetzung und Verifizierung von Bartok trauen müsse, inklusive dem sicheren Generieren des x86 Codes. Man arbeite daher an einer *TAL*[8], einer typsicheren Assemblersprache, um den Output des Compiler zusätzlich verifizieren zu können und später Bartok in der TCB durch einen kleineren, wesentlich einfacheren Verifizierer ersetzen zu können.

### 7. Garbage Kollektoren - Viele Köchen bereiten einen besseren Brei zu

Garbage Kollektoren sind essentiell für die meisten typ/speichersicheren Sprachen. Probleme bei den heutigen Umsetzungen existieren dabei, dass diese Algorithmen meist nicht effektiv für alle Anwendungen agieren. In Singularity werden Kernel und Prozess Adressräume von Garbage Kollektoren gesäubert. Durch eine vollständige Isolation von Prozessen ist es möglich mehrere

[5] *Translation Lookaside Buffer*

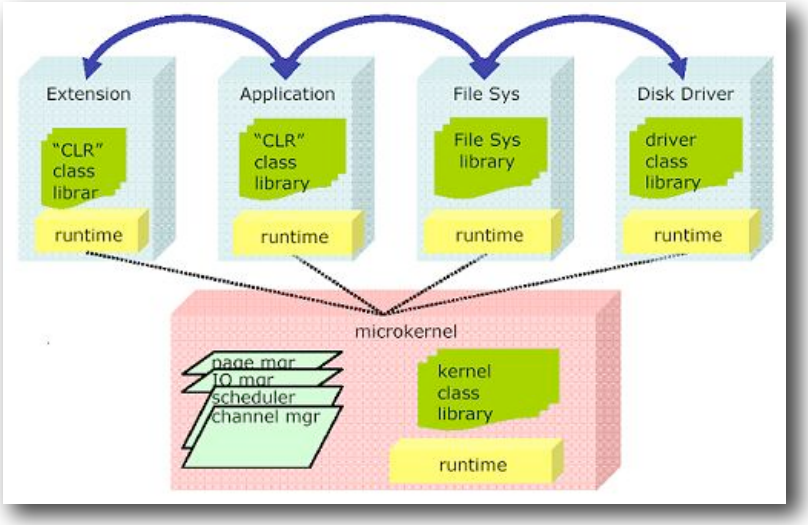
[6] *Trusted Computing Base*

[7] *Hardware Abstraction Layer*

[8] Typed Assembly Language

Kollektoren für verschiedene Anwendungen einzusetzen. Diese benötigen insbesondere keine globale Koordination. Ein weiterer Grund besteht darin, dass Zeiger nicht in andere Adressräume zeigen dürfen. Dadurch wird es möglich durch verschiedene Algorithmen eine effektivere Säuberung von Speicherbereichen zu erreichen. [HAF+07]

## 8. Architektur und Innenleben




[HLA+05]: Singularity architecture.

Werfen wir nun einen Blick in Singularity hinein. Programme bestehen hier nämlich aus einem Manifest und einer Ansammlung von Ressourcen. Das Manifest ist, wie beispielsweise in .NET, eine XML Datei, welche die SIPs beschreibt, die zusammen ein Programm ausmachen. Das Manifest wird hauptsächlich aus autogenerierten Metadaten vom Code erstellt. Es beschreibt die Programme in ihren Abhänigkeiten und in ihren Ressourcen, die sie zur Ausführung benötigen. Weiterhin enthält es „deklarative Statements", welche den gewünschten Endzustand einer Applikation nach der Installation oder dem Update angeben. Wenn man ein Programm in Singularity starten möchte, startet man als allererstes das Manifest. [HLA+05]

[LH08] fügt ein weiteres Architekturmerkmal ein: 95% des Kernels wurde in C# geschrieben. Davon sind 17% „unsicheres" C# und 5% C++ oder Assembler Code. Dabei ist anzumerken, dass aller User-Code verifizierbar sicher ist.
Alle Prozesse nutzen den selben virtuellen Adressraum. Dabei wird dieser virtuelle Speicher aufgeteilt in den Kernel Objekt Raum, den Objektraum für jeden einzelnen Prozess und den noch später zu behandelnden Exchange Heap. Mehr Speicher kann dabei ein Prozess bekommen, indem er den Kernel Page Manager bittet um neue, noch nicht vergebene Seiten. Diese müssen nicht fortlaufend sein, da die Garbage Collection dies nicht notwendigerweise fordert. Dabei müssen sich Prozesse im schlechtesten Fall auf einer 32-bit Architektur 4 Gigabyte[9] virtuellen Speicher teilen. Erweitern lässt sich dies durch eine 64-bit Architektur beispielsweise. [HLA+05]

[9] Dabei entspricht eine virtuelle Adresse einer Breite von 32 Bit, zweimal je 10 Bit für eine zweistufige Seitentabelle und 12 Bit (entspricht 4 Kilobyte) für den Offset. Somit lassen sich $2^{10} \times 2^{10} \times 2^{12}$ Byte adressieren. Das entspricht $2^{32}$ Byte, also 4 Gigabyte.

## 9. Umsetzung von Isolationsprinzipien in Singularity

In Singularity werden die Isolationsprinzipien in den softwareisolierten Prozessen umgesetzt. SIPs sind unprivilegierte Prozesse. Jeder einzelne besitzt eine eigene softwareisolierte Domäne, wodurch dieses Konzept viel billiger ist als eine Umsetzung in Hardware. Das heisst jeder SIP läuft in einem eigenen geschlossenen und komplett isolierten Prozessraum, wie es von der State Isolation Invariant gefordert wird. Desweiteren ist es einem SIP nicht erlaubt Referenzen auf Objekte anderer zu besitzen. Insbesondere ist der Zugriff auf fremde Objekte über Zeigerarithmetik, Cast-Operationen oder mittels privilegierter Instruktionen ausgeschlossen.
Dem Kernel sind allerdings weiterhin die Ausführung privilegierter Operationen vorbehalten. Die Durchsetzung erfolgt durch die statische Überprüfung auf Typ- und Speichersicherheit zur Kompilierung.
Weitere privilegierte Operationen werden durch Systemrufe exportiert. Dabei ist darauf zu achten, dass der Kernel keine Systemrufe bereitstellt, die die Closed API Invariant brechen könnten. [WYA+07]

## 10. Die Sache mit den Treibern und den Debuggern

In Singularity laufen Treiber für sich in eigenen softwareisolierten Prozessen ab. Dadurch ist es möglich eine vollständige Isolation durchzusetzen. In Zahlen gesprochen sind ca. 85% der heutigen Windows-Abstürze auf fehlerhafte Treiberprogrammierungen zurückzuführen [Bal08]. Es ist daher ein kluger Ansatz prinzipiell die Treiber aus dem Kernel auszulagern. Andrew Tanenbaum schreibt in [THB06], dass eine Mikrokernelarchitektur (wie Singularity) weniger fehleranfällig wäre, weil sie einfach weniger Code umfasse und die meisten Teile in das Userland ausgelagert wären. Fehlerhafte Treiber könnten somit keinen direkten Schaden am Kernel anrichten und würden durch ihre Auslagerung direkt zur Stabilität des Gesamtsystems beitragen.
Linus Torvalds kommentierte diese Behauptungen mit: *„Jeder, der sich mit der Programmierung von verteilten Systemen auseinander gesetzt hat, sollte wissen, dass wenn nur ein Node ausfällt, oftmals auch der Rest ausfällt"*. Er hält es daher für eine unzureichende Aussage, dass die bloße Treiberauslagerung zur Stabilität beitragen würde. Weiterhin schrieb er, dass auch ein Treibercrash in einem monolithischen System sich nicht gleich zum Gesamtcrash ausbreiten würde. [Tor06]

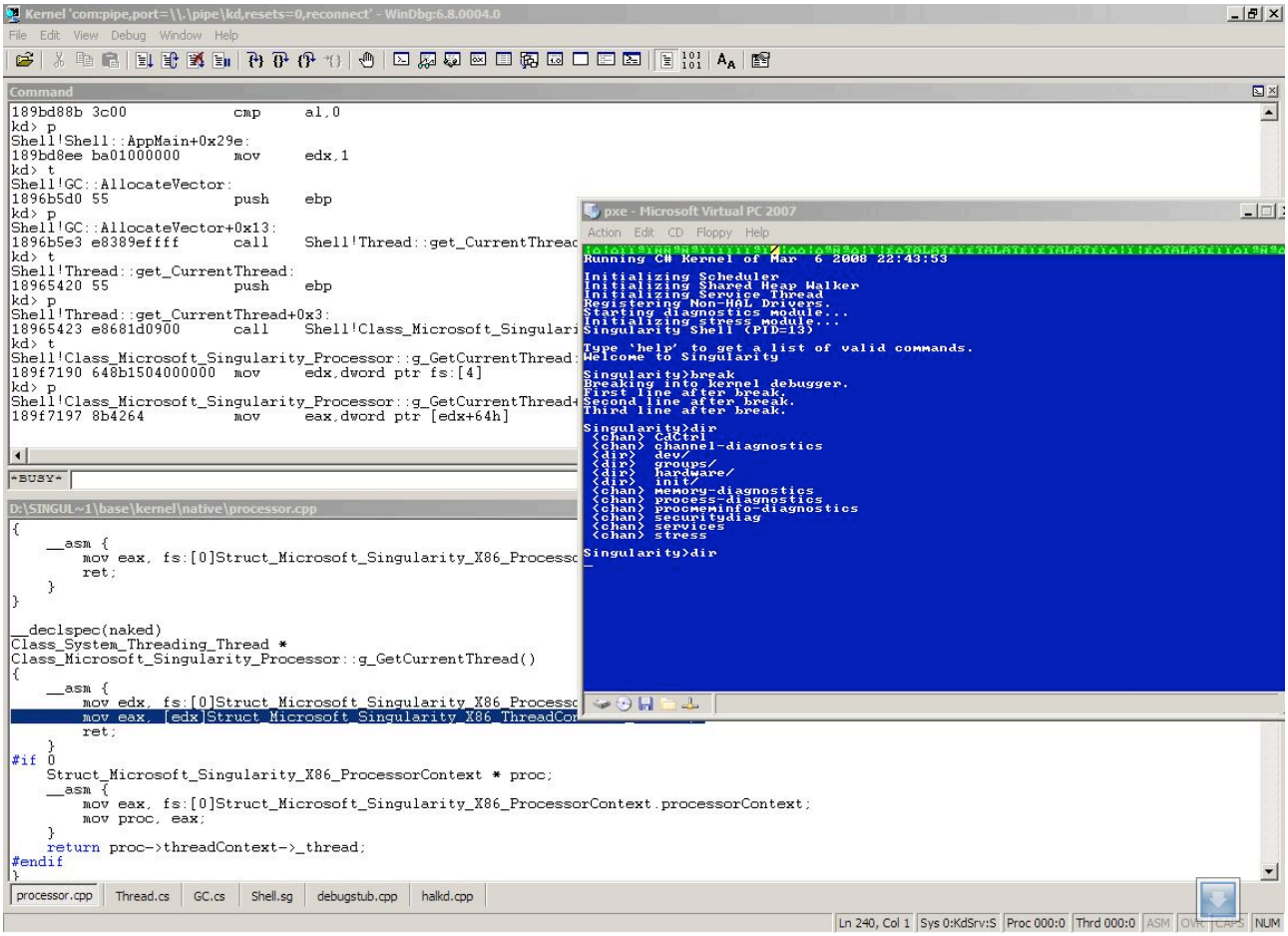
Ein weiteres Problem bei Treibern existiert dabei, dass es „normalen" Prozessen in Singularity nicht erlaubt ist Maschinencode auszuführen. Das Erlauben hätte zur Folge, dass beliebige Prozesse die Speichersicherheit verletzten könnten, sowie Speicherbereiche anderer Prozesse direkt über- oder neugeschrieben werden könnten. Treiber allerdings benötigen die Möglichkeit mit Maschinencode Zustände der (externen) Hardware verändern zu dürfen.
Eine Möglichkeit dies zu realisieren, wäre, wie andere 3rd-party Applikationen, auch Treiber als sichere MSIL-Binarys Singularity vorzulegen. Dadurch könnte eine statische Verifizierung durch Bartok weiterhin erfolgen. Zusätzlich jedoch müssten Treiber ein Zertifikat von ihrem Hersteller mitbringen, das belegt, dass der Treiber nach gutem Wissen und Gewissen programmiert worden ist. Durch diese Treiberzertifizierung wäre es möglich ihnen eine Maschinencode-Ausführung dennoch zu gestatten. Das Konzept des Signierens von Treibern ist im Übrigen nicht neu. Es wird bereits jetzt in Windows XP und Vista durchgeführt.[10]

---

[10] *Device Management and Installation Step-by-Step Guide: Signing and Staging Device Drivers.* http://technet.microsoft.com/en-us/library/cc754052.aspx

Bei den Debuggern liegen diese Probleme ebenfalls auf der Hand. Das Steuern und Inspizieren von zu untersuchendem Programmcode wird dabei wenig Probleme machen. Jedoch ist das Modifizieren von Hauptspeicherregionen, Ein/Ausgabe-Zuständen oder Registerzuständen der CPU von separierten Prozessen den „normalen" SIPs nicht gestattet, da es die State Isolation Invariant verletzen würde. Es ist keinem Prozess gestattet, von außen einen Prozess so zu modifizieren, dass er in einen anderen Zustand gelangt. Dies ist für Debugger unter Singularity wohl ein großes Problem.
Eine theoretische Möglichkeit wäre es Debuggern keinen Zugriff auf Maschinencode zu geben, jedoch auf das Modifizieren der MSIL-Binarys, der zu untersuchenden Programme. Dadurch erhielte man weiterhin die Möglichkeit zur Zeit der Übersetzung einen Verifizierer Überprüfungen auf Typ- und Speichersicherheit sowohl auf Einhaltungen der Isolationseigenschaften zu gestatten.

Praktischerweise hält Singularity selbst ein wenig C-Code im Kernel vor [HLA+05]. Dies tut es zum Beispiel zum größten Teil fürs Debugging. Dabei bietet es selbst keinerlei Werkzeuge hierzu an. Jedoch besitzt das Hostsystem (Windows) der virtuellen Maschinen diverse Werkzeuge. Diese können dazu benutzt werden um Singularity und alle in ihm laufenden Programme (SIPs) näher zu untersuchen, wie folgendes Bild illustrieren soll:




[11] Bildquelle: Ahmed Essam. Singularity Step by Step. http://www.ahmed-essam.com/2008/03/preparing-building-and-denuging.html. 2008

## 11. Kommunikation in vollständiger Isolation

Wie wir sahen, ist eine „normale“ Kommunikation via Shared Memory oder mittels der Übergabe von Objektreferenzen in Singularity nicht möglich durch die strickte Prozessisolation. Daher benötigt man andere Wege, die trotzdem die Kommunikationsinvarianz durchsetzen. Als weitere Forderung stellte man sich das performante Kommunizieren von großen Datenmengen mit möglichst keinem Kopieraufwand. Dabei ist anzumerken, dass Kommunikation ohne Kopieren auf herkömmlichen Systemen wohl gar nicht möglich ist. Erreichen kann man dies nur, wenn man sich auf die Durchsetzung der Typ- und Speichersicherheit durch den Compiler komplett verlassen kann.

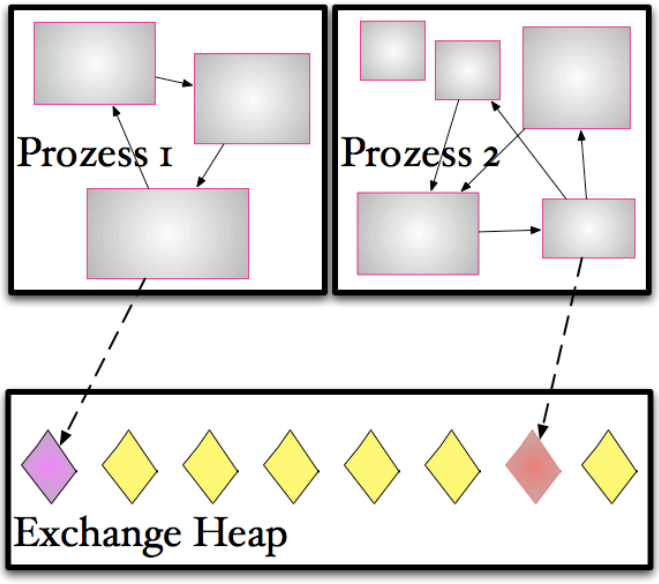
### 11.1 Aufgabe des Exchange Heaps während der Kommunikation

Wie in [FAH+06] beschrieben, ist der Exchange Heap eine Region im Hauptspeicher, die von Singularity speziell für die Kommunikation genutzt wird. Sie beinhaltet Nachrichten und Endpunkte. Jedoch ist es SIPs nicht erlaubt Objekte oder Referenzen auf Objekte direkt zu kommunizieren. Es sind daher lediglich primitive Datentypen oder Strukturen erlaubt ausgetauscht zu werden. Dabei sind jedoch Verweise in den Exchange Heap selbst gestattet. So ist es den SIPs möglich auch komplexere Nachrichtenstrukturen auszutauschen. Dabei hat jedes Datum zu jeder Zeit einen eindeutigen Besitzer. Wobei ein Prozess oder ein Thread Besitzer sein kann.

Die Garbage Kollektoren räumen jedoch diese Region nicht mit auf. Deshalb verwendet man hier Referenzzähler, um die Benutzung von Blöcken zu verfolgen. [HAF+07]

Im Exchange Heap befinden sich weiterhin die Nachrichten-Queue und die Signalvariablen, die dem Empfänger das Anliegen einer Nachricht zu erkennen geben sollen.

Die Grafik stellt die Verweis-Relation dar. So darf jeder Prozess Verweise (etwa Pointer) in sich selbst besitzen. Weiterhin stellt man fest, dass der Exchange Heap lediglich Pointer in sich selbst besitzt, jedoch niemals Pointer nach außen.
Und da jeder Speicherbereich im Exchange Heap exakt einem Besitzer untersteht, setzt man implizit das *mutual exclusion*[12] Prinzip durch. [HLA+05]




Wirft man einen Blick auf die Fehlertoleranz so stellt man fest, dass eine Beendigung eines fehlerhaften Prozesses samt Freigabe von Ressourcen sehr schwierig ist beim herkömmlichen Shared Memory Prinzip. Desweiteren kann dies zu inkonsistenten Zuständen der Kommunikationspartner führen. Java löst dieses Problem beispielsweise durch die Terminierung aller Teilnehmer einer Kommunikation.
Eine sehr elegante Lösung sind daher die SIPs, da sie sich keinerlei Speicher miteinander teilen. Eine

[12] Prinzip vom gegenseitigen Ausschluss

Terminierung ist daher immer möglich. Bei einer solchen bekommen die übrigen Teilnehmer eine Exception vom Kernel geworfen. Sie haben sich dann jedoch lediglich mit lokalen Begebenheiten auseinander zu setzen. Eine Terminierung von allen Partnern ist daher nicht mehr nötig. [FAH+06]

### 11.2 Ein Kanal für den gegenseitigen Austausch

Um Kommunikation zwischen Prozessen zu ermöglichen, führten die Singularity-Entwickler in Sing# das Prinzip der *Channels* ein.
Dabei bilden Channels einen bidirektionalen Nachrichtenfluss, wobei Nachrichten „in order" zugestellt werden. Sie bestehen aus zwei Endpunkten, welche, wie beschrieben, im Exchange Heap liegen. Dabei sind jeweils zwei symmetrisch zu einander. Man nennt einen Endpunkt *Exporter* (`C.Exp`) und den anderen *Importer* (`C.Imp`). `C` geht dabei auf den Channel *Contract* zurück. Wenn man den Vergleich zum Klient-Server-Model wagt, so ist der Exporter eher der Server und Importer eher der Klient. Diese Endpunkte liegen als Datenstruktur vor und besitzen jeweils eine eigene Empfangswarteschlange. Außerdem können sie auch selbst über einen Channel kommuniziert werden. Dadurch bewahrt man sich die Möglichkeit ein dynamisch wachsendes Kommunikationsnetzwerk aufzubauen. [HLA+05]

### 11.3 Es wird ein Vertrag geschlossen, an den sich alle halten

Contracts bilden das Fundament der gemeinsamen Kommunikation. Sie bezeichnen Kommunikationsverträge zwischen den jeweiligen Partnern. Dabei ist darauf zu achten, dass die Details so explizit wie möglich beschrieben werden. Wobei der Entwurf immer aus Perspektive des Exporters beschrieben wird. Der Importer verhält sich dabei komplementär.
Obligatorische Angaben in einem Contract wären zum Beispiel der Nachrichtentyp, die Nachrichtenargumente, sowie das eigentliche Kommunikationsprotokoll. Dies beschreibt die Reihenfolge der Nachrichten und die Aktionen bei eingehenden/ausgehenden Nachrichten. Außerdem wäre unter anderem die Richtung der Nachrichten optional. Dabei gilt standardmäßig eine bidirektionale Richtung.
Das Beschreibungsmittel für die Contracts bilden endliche Automaten. Das heißt man hat wohldefinierte Übergänge von einem Zustand in einen anderen. „Unbeschriftete" Kanten darf es daher nicht geben. Zu jeder Aktion gibt es eine oder mehrere gültige Reaktionen.
Die Verträge werden durch Bartok zur Kompilierzeit auf Korrektheit und Einhaltung überprüft. Einen Verstoß würde der Compiler daher sofort ahnden. [HLT+05] bezeichnet dieses Verfahren als *conformance checking*. Dies beschreibt ein Verfahren, welches sicherstellt, dass der Contract umfassend beschrieben worden ist. Das heißt, dass kein möglicher Übergang existiert, wenn es im endlichen Automaten keine Kante gibt. Desweiteren wird sichergestellt, dass zwei Kommunikationspartner durch den Contract nicht in einem Deadlock enden, sowie dass niemand eine Nachricht erhält, die er nicht ausdrücklich erwartet.
Zusätzlich wird die Zustandsisolation überprüft werden. Das heißt, dass die expliziten Regeln, in Bezug auf Kommunizieren von Objekten und so weiter, strickt durchgesetzt werden. [FAH+06]

### 11.4 Konkrete Umsetzung der Kommunikation in Singularity

Zu allererst schaue man sich das Senden von Botschaften an. Dieses geschieht nach [HLA+05] nicht blockierend. Der Sender wird Speicher im Exchange Heap reservieren mittels Systemruf. Das heißt er erhält (einen) Zeiger auf die Nachrichteninhalte in dieser Hauptspeicherregion. Die Variablen, die allokiert werden sollen, müssen explizit im Programmcode gekennzeichnet werden, damit sie von Daten auf privaten Heaps eines SIPs unterschieden werden können. Hierbei ist anzumerken, dass es sich dabei um keine Zeiger, die auf interne Strukturen eines SIPs zeigen, handeln darf, die ein SIP versucht zu kommunizieren. Als zweites wird ein Zeiger auf den Nachrichteninhalt als Argument an den Message-Typ angehangen werden. Dieser wurde vorher im Contract definiert. Danach legt der Sender die zu versendende Nachricht in die Empfangsqueue des Empfängers. Dabei kennt der Sender den Zeiger des Channelendpunkts des Empfängers. Als letzten Schritt signalisiert der Sender dem Kernel, dass die Nachricht für den Empfänger vorliegt. Daraufhin wird dieser vom Scheduler aufgeweckt werden.

Das Empfangen von Nachrichten geschieht synchron, das bedeutet in Reihenfolge der Absendung, wenn nichts anderes im Contract beschrieben worden ist. Der Empfänger prüft hierbei ob die zu erwartende Nachricht am Anfang der Queue für ihn vorliegt. Ist dies nicht der Fall, blockiert er und wird daher vom Scheduler wieder schlafen gelegt bis die entsprechende Nachricht eintrifft. Ist die Nachricht in der Queue, aber nicht am erwarteten Anfang, so wird ein Fehler gemeldet und der Kernel schreibt in den Channel die `ChannelClosed`-Nachricht.

In diesem Zusammenhang sollte man sich verdeutlichen wann eine Kommunikation beendet ist, beziehungsweise wann ein Channel geschlossen werden kann. Wenn ein Prozess normal den Austausch von Botschaften beenden möchte, beziehungsweise normal terminiert, wird er ein `delete ep` aufrufen. Dann, oder wenn er abrupt terminiert, wird der Kernel den Endpunkt widerrufen. Dabei ist anzumerken, dass generell die beiden Endpunkte unabhänig voneinander geschlossen werden. Dies geschieht für eine saubere Fehlersemantik. Desweiteren wird danach der Channelspeicher wieder freigegeben. Danach schreibt der Kernel das angesprochene `ChannelClosed` in die Empfänger-Queue. Der Compiler hat dann dafür Sorge zu tragen, dass kein Senden oder Empfangen mehr möglich ist.

Diese Art der Kommunikation untergräbt prinzipiell die geforderten Isolationseigenschaften, könnte man behaupten, da man den SIPs doch eine manuelle Verwaltung von Zeigern für die Kommunikation zugesteht. Daraus könnten *Dangling Pointer* entstehen - beispielsweise vom Sender auf Nachrichtenargumente und freigegebene/undefinierte Speicherbereiche nach dem Übertrag an den Empfänger. Dabei gilt jedoch, dass in jedem Fall der Contract eingehalten werden muss, was die statische Überwachung vom Compiler garantiert. Weiterhin gilt, dass jeder Prozess nur auf die Bereiche im Exchange Heap zu greifen darf, die er selbst in diesem Moment besitzt.

Als Beispiel für eine Kommunikation in Singularity ist das Versenden einer Datei von der Festplatte zu einem entfernten Rechner zu nennen. Dabei werden Netzwerkpakete und die Diskblöcke zwischen den Treibern, dem Netzwerkstack und dem Dateisystem als Botschaften ausgetauscht mittels Channelkonzept. [FAH+06]

## 12. Zahltag - Zahlen und Fakten

Wie man sah, ist Singularity besonders bei Kontextwechsel sehr sparsam, da diese inklusive Kopieraufwand und TLB flushen komplett wegfallen. Alle Prozesse nutzen dieselbe einheitlich systemweite Pagetabelle.
Microsoft selbst beziffert in [Duf06] einen Kontextwechsel mit 2.000 bis 8.000 Prozessorzyklen, eine Windows-Thread-Erstellung mit 200.000 und eine Löschung mit 100.00 Zyklen. Wie Felix von Leitner in [Lei02] schrieb, ist dabei die Betrachtung einer x86 Architektur jedoch schon großzügig, da diese wenige Register besitzt, welche man im Falle eines Kontextswitchtes sichern und laden muss. Auf einer *ARM*[13]-Architektur muss man beispielsweise zusätzlich die Pipeline leeren und die logischen Caches flushen. Der Heise Verlag beziffert in [Sti05] weiterhin eine Prozesserstellung in Singularity mit 300.000, bei Windows mit 5,4 Millionen, bei einem BSD mit einer Million und bei Linux mit 719.000 Zyklen der CPU.

| | API Aufruf | Message Ping/Pong | Prozesserstellung |
|---|---|---|---|
| Singularity | 80 | 1.041 | 388.162 |
| FreeBSD | 878 | 13.304 | 1.032.254 |
| Linux | 437 | 5.797 | 719.447 |
| Windows | 627 | 6.344 | 5.375.735 |

[HAF+07] stellte ein Benchmarking von Singularity auf. Dabei wurde es mit FreeBSD, Windows und Linux verglichen auf einem AMD Athlon 64 3000+ (1.8 GHz) mit 1 GB RAM.
Alle Werte sind Prozessorzyklen auf dem jeweiligen Betriebssystem, die gebraucht wurden, um folgende Aufgaben zu erfüllen:

Zuerst verglich man den Aufruf eines Systemrufs. Auf den Unix-Systemen war dies `clock_getres()` auf Windows `SetFilePointer()` und auf Singularity `ProcessService.GetCyclesPerSecond()`. Sehr deutlich sieht man hier, dass Singularity enorme Pluspunkte sammelt durch das Einsparen des Wechsels zwischen „Kernel Mode“ und normalem Ausführungsmodus (in einem höheren Ring). Dadurch erreicht man eine fünf bis zehn mal schnellere Ausführung als in den restlichen Betriebssystemen.

Beim Message Ping/Pong schob man ein Byte große Nachrichten zwischen zwei Prozessen hin und her. Dabei verwendete man auf dem Windows NamedPipes, auf den Unix-Systemen Sockets und in Singularity Channels. Auch hier machen sich wieder Vorteile durch die Isolation in Software in

[13] *Acorn Risc Machine*

Singularity bemerkbar, zum Beispiel, dass Sender und Empfänger im gleichen Adressraum sich befinden. Jedoch kommt ein weiterer Vorteil, dass Einsparen des Kopieraufwandes bei der Kommunikation gegenüber herkömmlichen Methoden, kaum zum Tragen, weil ein Byte relativ schnell kopiert werden kann.

Bei der Prozesserstellung sticht vor allem das Windows durch eine extrem schlechte Performanz hervor. Dies lässt sich damit begründen, dass bei einem `fork()` eben nicht wie in Unix-Systemen üblich vom Vaterprozess vererbt wird. Singularity erstellt einen Prozess circa zwei bis achtzehn mal schneller. Dies könnte wieder an den extrem billigen API-Calls liegen, sowie an der systemweit einheitlichen Pagetabelle.

## 13. Zusammenfassung

Singularity setzt Isolationsprinzipien neu um. Anders als die in dieser Themenreihe vorgestellten Konzepte von *Asbestos*, *L4/Nizza*[Neu09] oder *SELinux* baut es ausschließlich auf die Isolierung durch Software. Die eingeführten SIPs besitzen einzelne softwareisolierte Domänen. Es braucht dadurch nichtmehr die gewöhnlichen Hardwaremechanismen wie etwa CPU-Ringe oder getrennte virtuelle Seiten durch die MMU. Zusätzlich dazu setzt der Compiler eine statische Überprüfung des Bytes-Zwischencodes durch. Dies vermeidet gewöhnliche Fehler wie Pufferüberläufe oder andere Speicherfehler. Die Entwickler von Singularity gaben mit Sing# dem alten Konzept des Managed Codes ein neues Gesicht.

Man kann nur hoffen, dass dieses Forschungsbetriebssystem kein schwarzes Loch der Ideen wird, sondern diese Konzepte auch in unsere heutigen Betriebssysteme Einhalt finden werden.

Ein anderes denkbares Szenario wären beispielsweise unsere heutigen Smartcards. Dort kommt es besonders auf die Größe des Prozessors an. Wenn man hierbei die Hardwareunterstützung für die Umsetzung der Isolationsprinzipien minimieren würde, durch die Idee der SIPs, könnte man Transistoren auf dem Prozessor Die[14] einsparen beziehungsweise anderweitig verwenden.

Am 14. November 2008 wurde Singularity 2.0 veröffentlicht.

[14] Begriff aus der Halbleitertechnik - etwa „Nacktchip“

# ZITIERTE WERKE


**[Bal08]**: Thomas Ball. ***The Verified Software Challenge: A Call for a Holistic Approach to Reliability***, p. 43. Springer, Berlin/Heidelberg, 2008. http://www.springerlink.com/content/u152u5601416m047/

**[Duf06]**: Joe Duffy. ***Verwenden der Parallelität für Skalierbarkeit***, 2006. http://msdn.microsoft.com/de-de/library/cc749798.aspx

**[Ess06]**: esser. ***Singularity***. Wiki der Humboldt Universität für Informatik, 2006. Revision 6525. http://sarwiki.informatik.hu-berlin.de/Singularity

**[FAH+06]**: Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, Steven Levi. ***Language support for fast and reliable message-based communication in singularity OS***. In Proc. of the 2006 EuroSys Conference, pages 177-190. ACM SIGOPS, March 2006. http://www.cs.kuleuven.ac.be/conference/EuroSys2006/papers/p177-fahndrich.pdf

**[HAF+07]**: Galen Hunt, Mark Aiken, Paul Barham, Manuel Fähndrich, Orion Hodson, James Larus, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, Brian Zill. ***Sealing OS Processes to Improve Dependability and Security***. Microsoft Research Technical Report MSR-TR-2005-51, Apr. 2006. http://research.microsoft.com/research/pubs/view.aspx?type=Technical%20Report&id=1095

**[HL04]**: Galen C. Hunt, James R. Larus. ***Singularity Design Motivation***. no. MSR-TR-2004-105, pp. 4, Microsoft Research, Nov. 2004. http://research.microsoft.com/apps/pubs/default.aspx?id=70101

**[HL07]**: Galen Hunt and James Larus. ***Singularity: Rethinking the Software Stack***. Operating Systems Review. Vol. 41, Iss. 2, pp. 37-49, April 2007. ACM SIGOPS. http://research.microsoft.com/apps/pubs/?id=69431

**[HLA+05]**: Galen Hunt, James R. Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, Brian D. Zill. ***An Overview of the Singularity Project***. Microsoft Research Technical Report MSR-TR-2005-135, Oct. 2005. http://www.research.microsoft.com/research/pubs/view.aspx?type=Technical%20Report&id=989

**[HLT+05]**: Galen Hunt, James Larus, David Tarditi, and Ted Wobber. ***Broad New OS Research: Challenges and Opportunities***. USENIX Tenth Workshop on Hot Topics in Operating Systems (HOTOS X), June 2005. http://www.usenix.org/events/hotos05/final_papers/hunt.html

**[Lei02]**: Felix von Leitner. ***Skalierbare Netzwerkprogrammierung***, 2002. http://www.fefe.de/scalable-networking.pdf

**[LH08]**: James Larus, Galen Hunt. ***Using the Singularity Research Development Kit (RDK)***. Tutorial, 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008) Slides, Seattle, WA, March 1, 2008. http://research.microsoft.com/en-us/projects/singularity/asplos2008_singularity_rdk_tutorial.pdf

**[LHT]**: James Larus, Galen Hunt, and David Tarditi. ***{End Bracket} Singularity***. MSDN Magazine, Vol. 21, No. 7, pp. 176. http://msdn.microsoft.com/de-de/magazine/cc163603(en-us).aspx

**[Neu09]**: Felix Neumann. ***Die Nizza-Systemarchitektur***, 2009.
http://www.the-contented.de/DieNizzaSystemarchitektur

**[Sti05]**: Andreas Stiller. ***Microsofts verlässliches Betriebssystem "Singularity"***. heise online, 2005.
http://www.heise.de/newsticker/Microsofts-verlaessliches-Betriebssystem-Singularity--/meldung/65804

**[THB06]**: Andrew S. Tanenbaum, Jorrit N. Herder, Herbert Bos. ***Can We Make Operating Systems Reliable and Secure?***, Vrije Universiteit, Amsterdam, 2006.
http://www.computer.org/portal/site/computer/menuitem.5d61c1d591162e4b0ef1bd108bcd45f3/index.jsp?&pName=computer_level1_article&TheCat=1005&path=computer/homepage/0506&file=cover1.xml&xsl=article.xsl

**[Tor06]**: Linus Torvalds. ***Hybrid kernel, not NT***, 9.5.2006.
http://www.realworldtech.com/forums/index.cfm?action=detail&id=66630&threadid=66595&roomid=2

**[WP05+]**: Autorenkollektiv. ***Singularity***. Wikipedia, 2005, 2006, 2007, 2008. Revision 54484656.
http://de.wikipedia.org/wiki/Singularity

**[WYA+07]**: Ted Wobber, Martín Abadi, Andrew Birrell, Dan Simon, and Aydan Yumerefendi. ***Authorizing applications in Singularity***. In European Conference on Computer Systems (EuroSys), Lisbon, Portugal, March 21-23, 2007. http://research.microsoft.com/apps/pubs/default.aspx?id=59976

## LIZENZTEXT