Operating Systems Engineering

“Singularity”

Experiment in language/OS co-design

By Dan Tsafrir, 24/5/2017.
Largely based on “Singularity: Rethinking the Software Stack”
OSR, April 2007, http://dx.doi.org/10.1145/1243418.1243424
Singularity

- An experimental research OS

- By Microsoft Research Operating Systems Group
People

- **Leads**: Galen Hunt, Jim Larus

- **Members**: Mark Aiken, Paul Barham, Richard Black, Trishul Chilimbi, Chris Hawblitzel, John DeTreville, Ulfar Erlingsson, Manuel Fähndrich, Wolfgang Grieskamp, Tim Harris, Orion Hodson, Rebecca Isaacs, Mike Jones, Steven Levi, Roy Levin, Nick Murphy, Dushyanth Narayanan, Sriram Rajamani, Jakob Rehof, Wolfram Schulte, Dan Simon, Bjarne Steensgaard, David Tarditi, Ted Wobber, Brian Zill, Ben Zorn

- **Interns**:
  - **2007**: Ryan Braud (University of California, San Diego), Michael Carbin (MIT), Michael Emmi (UCLA), Gabriel Kliot (Technion - Israel Institute of Technology), Ross McIlroy (University of Glasgow), Filip Pizlo (Purdue University), Polvios Pratikakis (Univ. of Maryland)
  - **2006**: Marc Eaddy (Columbia University), Haryadi S. Gunawi (Univ. of Wisconsin - Madison), Hiroo Ishikawa (Waseda University), Virendra J. Marathe (Rochester), Polvios Pratikakis (University of Maryland), Roussi Roussev (Florida Tech), César Spessot (Universidad Tecnológica Nacional Facultad Córdoba)
  - **2005**: Michael Carbin (Stanford), Adam Chlipala (UC Berkeley), Martin Pohlack (TU Dresden), Avi Shinnar (Harvard), Mike Spear (Rochester), Aydan Yumerefendi (Duke)
  - **2004**: Jeremy Condit (UC Berkeley), Daniel Frampton (Australian National University), Chip Killian (UC San Diego), Fernando Castor de Lima Filho (Universidade Estadual de Campinas), Prince Mahajan (IIT Roorkee), Bill McCloskey (UC Berkeley), Martin Murray, Tom Roeder (Cornell), Avi Shinnar (Columbia), Yaron Weinsberg (Hebrew University of Jerusalem)
Singularity

- Developed since 2003
  - Open source since 2008
  - (Last release: Nov 2008)
  - Can be freely used by academia

- A few publications in good places (last one in 2009)
  - Not as many as you’d expect, given a project that size

- Lots of people worked on it; lots of $s were spent
Motivation

❖ Official
  ❖ Rethink SW stack, as many decisions are from late 1960s &
    early 1970s (in all OSes they’ve stayed ~ the same)

❖ Speculated:
  ❖ Windows isn’t the best OS you could imagine…
    (UNIX-like aren’t that different…)
  ❖ Microsoft is searching for ways to improve
Goals

◆ Answer the question
  ❖ “What would a SW platform look like if it was designed from scratch with the primary goal of improved dependability and trustworthiness”?

◆ Attempt to eliminate the following obvious shortcomings of existing systems & SW stacks
  ❖ Widespread security vulnerabilities
  ❖ Unexpected interactions among applications
  ❖ Failure caused by errant
    ▪ Extensions
    ▪ Plug-ins, add-ons
    ▪ Drivers
  ❖ Perceived lack of robustness
Strategies to achieve goals

1. **Pervasively employing safe programming language**
   - Kernel (where possible) & user space
   - Eliminates such defects as buffer overflow
   - Allows for better reasoning about the code

2. **Make it possible/easier to use program verification tools**
   - Impose constraints to make verification feasible
   - May guaranty entire classes of programming errors are eliminated early in the development cycle

3. **Improved system architecture & design**
   - Stops propagation of runtime errors at well-defined boundaries
   - => Easier to achieve robust & correct behavior
Kernel

- **Microkernel structure**
  - Factored services into user space: NIC, TCP/IP, …
  - Still, 192 sys call! (“may appear shockingly large”)
  - But actually, much simpler than number suggests
    - No sys calls with complex semantics (such as ioctl)
  - No UNIX/win compatibility to avoid their pitfalls (e.g., TOCTTOU)

- **Mostly written in Sing#: type-safe, garbage-collected language**
  - 90% of kernel written in Sing# (extension of C#)
  - 6% C++
  - 4% C, plus little assembly snippets

- **Significant “unsafe” parts, but still much better than alternative**
  - Garbage collector (in “unsafe” Sing#: accounts for 1/2 of unsafe code); memory management; I/O access subsystems
Kernel – cont.

◆ Provides 3 base abstractions (the architectural foundation)
  - Software-isolated processes (SIPs)
  - Contract-based channels
  - Manifest-based programs (MBPs)
Software-isolated processes (SIPs)

◆ Like an ordinary process,
  ❖ SIP is a holder of processing resources
  ❖ SIP provides context for program execution
  ❖ SIP is a container of threads

◆ One fundamental (somewhat surprising) difference
  ❖ The most radical part of the design…
  ❖ SIPs all run in *one* address space!
  ❖ Moreover, it’s the same address space as of the kernel
  ❖ And user code runs with full hardware privileges (CPL=0)!
Why might that be useful?

◆ Performance
  ❖ Fast process switching
    ▪ No page table to switch
    ▪ No need to invalidate TLBs
    ▪ In fact, can run in real mode (if memory \(\leq 4\)GB; singularity had 32bit address)
    ▪ In which case no need for TLB at all
  ❖ Fast system calls
    ▪ (CALL rather than INT 0x80, we’re at CPL=0 anyway)
  ❖ Fast IPC
    ▪ No need to copy (kernel \(\leftrightarrow\) user) or remap, it’s already in your address space
  ❖ Direct user-program access to HW
    ▪ E.g., device drivers of the microkernel
### Some performance numbers

<table>
<thead>
<tr>
<th></th>
<th>cost (in CPU cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>API (sys) call</td>
</tr>
<tr>
<td></td>
<td>thread yield</td>
</tr>
<tr>
<td></td>
<td>message ping/pong</td>
</tr>
<tr>
<td></td>
<td>process creation</td>
</tr>
<tr>
<td>Singularity</td>
<td>80</td>
</tr>
<tr>
<td>FreeBSD</td>
<td>(x11) 878</td>
</tr>
<tr>
<td>Linux</td>
<td>(x5.5) 437</td>
</tr>
<tr>
<td>Windows</td>
<td>(x7.8) 627</td>
</tr>
</tbody>
</table>

- Singularity
  - **API (sys) call**: 80
  - **thread yield**: 365
  - **message ping/pong**: 1,040
  - **process creation**: 38,800

- FreeBSD
  - **API (sys) call**: (x11) 878
  - **thread yield**: (x2.5) 911
  - **message ping/pong**: (x13) 13,300
  - **process creation**: (x27) 1,030,000

- Linux
  - **API (sys) call**: (x5.5) 437
  - **thread yield**: (x2.5) 906
  - **message ping/pong**: (x5.6) 5,800
  - **process creation**: (x19) 719,000

- Windows
  - **API (sys) call**: (x7.8) 627
  - **thread yield**: (x2.1) 753
  - **message ping/pong**: (x6.1) 6,340
  - **process creation**: (x139) 5,380,000

- **“context switch”**
- **IPC**
But we’re interested in robustness!

- Recall that performance is *not* the main goal
  - Rather, it’s robustness & security
  - Is *not* using page-table protection consistent with our goal?

- For starters, note that
  - Our OSes do use page tables and they’re not very robust…
  - Unreliability often comes from *unsafe lang.*, & *extensions*
    - Browser plug-ins & add-ons, loadable kernel modules,…
    - HW vmem protection irrelevant in any case for extensions

- Can we just do without HW protection?
  - If we can solve the problem of extensions, then yes!
  - It’d be the same solution for processes…
The idea

◆ Extensions (device drivers, new network protocol, plugins)
  ❖ Would employ a different process
  ❖ Communicate via explicit message passing IPC

◆ Challenges
  ❖ Prevent evil/buggy processes from writing
    ▪ To each other
    ▪ To the kernel
  ❖ Be able to cleanly kill/exit
    ▪ Avoid entangling memory spaces
    ▪ Would ease GC work

◆ The solution: SIPs
  ❖ Software-isolated processes
SIP philosophy: “sealed” (can’t be modified)

- No modifications from outside
  - No shared memory
  - No signals
  - Only explicit message-passing IPC

- No (code) modification from within
  - No JIT
  - No class loader
  - No dynamically loaded libraries
SIP rules

- **Only point to your own data (as in memory pointers)**
  - No pointers into other SIPs
  - No pointers into the Kernel
  - => No sharing despite shared address space!

- **SIP can allocate pages of memory from kernel**
  - Different (possibly consecutive) allocations might not be contiguous
Why un-modifiable?

- Why is it crucial that SIPs can’t be modified?
  - Can’t even modify themselves (code-wise)

- What are the benefits?
  - No straightforward code insertion attacks
    - E.g., `<script>..</script>` injected from the outside into a browser SIP will not work
    - (How, then, will an attack by a malicious site work?)
  - Easier to reason about (prove things) statically
  - Easier to optimize
    - E.g., inline (“whole program optimization” can often eliminate overheads of function virtualization in OOP)
    - E.g., delete unused functions

- Singularity vs. JVM (isn’t it “safe” too?…)
  - JVM can load classes in runtime
    - => Precludes previous slide
  - JVM allows its threads to share memory
    - => Entanglement for when killing
      (with singularity, cleanup is much simpler)
  - JVM allows locks et al to sync (!= explicit messages)
    - Typically impossible to reason about locks without knowing
      the program’s semantics; msgs make everything simpler
  - JVM = one runtime + one GC
    - But it has been shown that different programs can benefit
      from different GCs
    - Singularity supplies a few GCs (trusted code)

❖ Bottom line
  ❖ The improved performance reported earlier…
  ❖ …makes the pervasive use of isolation primitives like we describe feasible
Enforcing memory isolation

- **How to keep SIPs from reading/writing other SIPs?**
  - We must make sure that SIP exclusively reads/writes from/to memory the kernel has given to that SIP (and nothing else)

- **Shall we have compiler generate code to check every access?** (“Does this pointers points to memory the that kernel gave us?”)
  - Would slow code down significantly (especially since memory isn't contagious)
  - And actually, we don't even trust the compiler!
Enforcing memory isolation – cont.

- **Overall structure: during installation**
  - Compile to bytecode
  - Verify bytecode statically (confirms to SIP constraints + does not conflict with any other already existing SIP)
  - Compile bytecode to machine code
  - Later, run the verified machine code w/ trusted runtime

- **Verifier verifies SIP only uses reachable pointers**
  - SIP cannot create new pointers
    - Only trusted runtime can create pointers
  - Thus, if kernel/runtime never supply out-of-SIP pointer
    - => Verified SIP can use its memory only
Specifically, the verifier verifies that

- SIP doesn’t “cook up” pointers
  - Only uses pointers given to it (by the runtime)
- SIP doesn’t change its mind about type
  - E.g., don’t reinterpret int as pointer
- SIP doesn’t use pointer after it was freed
  - Enforced with (trusted) GC
- No other tricks exist
  - … 😊
Enforcing memory isolation – cont.

- **Bytecode verification seems to do *more* than Singularity needs**
  - E.g. cooking up pointers might be OK, as long as within SIP's memory
  - Thus, verifier might forbid some programs that might have been OK on Singularity

- **Benefits of full verification make it worth our while**
  - Fast execution, often don't need runtime checks at all
    - Though some still needed: array bounds, OO casts
  - Prove IPC contracts (a few slides down the road)
  - …
Trusted vs. Untrusted

- **Trusted SW**
  - If it has bugs => can crash Singularity or wreck other SIPs

- **Untrusted SW**
  - If it has bugs => can only wreck itself

- **Let’s consider some examples**
  - Compiler? (untrusted, we can have many of them)
  - Compiler output? (untrusted)
  - Verifier? (trusted)
  - Verifier output? (trusted)
  - GC? (trusted)
Price of SW & HW isolation

◆ WebFiles benchmark:
  ❖ I/O intensive (based on SPECweb99)
  ❖ Consists of three SIPs:
    1. Client – issues random static/dynamic file reads
       – Files by Zipf distribution, with sizes
         \[35\% \leq 1\text{KB} < 50\% \leq 10\text{KB} < 14\% \leq 100\text{KB} < 1\% \leq 1\text{MB} \]
    2. File system
    3. Disk device driver
  ❖ Times are all normalized against a default Singularity configuration
    ▪ SIPs run same address space
    ▪ Same privilege level as the kernel
    ▪ Paging HW off
IPC: contract-based channels

- Contract-based
  - Contract defines a state machine
- Verifier statically proves it at install time
  - Example: Listing 1 of
    - G. Hunt & J Larus, “Singularity: rethinking the SW stack” OSR 2007 (a survey on singularity)
    - Homework: read & understand it
- Very Fast
  - Zero copy (see slide 9)
    - SIP can only have one pointer to an object (verified)
  - Through the “exchange stack”
- Channels are “capabilities”
  - E.g., an open file is a channel received from the file server
  - If you got the channel, it means you’re allowed to access the file
Manifest-based programs (MBPs)

◆ No code is allowed to run without a “manifest”
  ▶ By default, a SIP can do nothing (has no power ⇔ no capabilities)
  ▶ Manifest describes SIP’s
    ▪ Capabilities, required resources, dependencies upon other programs
◆ When program/MBP installed, verifying that
  ▶ It meets all required safety properties
  ▶ All its dependencies are met
  ▶ Doesn’t interfere with already-installed MBPs
◆ Example
  ▶ Manifest of a driver provides enough information to prove it will not access HW of previously installed drivers
Aftermath: Midori

- Midori was a “secret” OS project in MS whose original goal is believed to be the commercialization of Singularity
  - http://en.wikipedia.org/wiki/Midori_(operating_system)

- Every once in a while we used to hear something about Midori in OS circles and the related media
  - “something seems to be afoot” [ZDNet, 29 Dec, 2013]

- Unofficially said to implemented in “M#”, which is supposed to be the successor of Sing#

- 2015: got the word it’s dead