Operating Systems Engineering

“Singularity”

Experiment in language/OS co-design

By Dan Tsafrir, 14/5/2014.
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
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… (Though 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 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 (Israeli connection: Yaron Weinsberg), file system, disk driver
  - Still, 192 sys call! (“my 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

- **Still, significant “unsafe” parts**
  - 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)
  - Manifest-based programs (MBPs)
  - Contract-based channels
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 <= 4GB)
      - 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 ↔ user), 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>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>

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 very 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 IPC
- The challenge?
  - Prevent evil/buggy processes from writing
    - To each other
    - To kernel
  - Be able to cleanly kill/exit
    - Avoid entangling memory spaces
    - Would allow GC to work
- The solution: SIPS
  - Software-isolated processes

SIP philosophy: “sealed”

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

- No 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…

◆ 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 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 nearly 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 to share memory
    - => Entanglement for when killing
      (With singularity cleanup is much simpler)
  - JVM allows other ways to sync (other than 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
  - 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 point to memory the 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
Enforcing memory isolation – cont.

- **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 interpret 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)
  - 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:
    - client – issues random file reads across files with a Zipf sizes
    - file system
    - 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
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 driver provides enough information to prove it will not access HW of previously installed drivers

Aftermath: Midori

- Midori is 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 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#