TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute for System Architecture, Operating Systems Group

Paper Reading: Singularity: Rethinking the Software Stack

Galen Hunt, James Larus
presented by Bjoern Doebel

2007-08-22

Motivation

"I liken starting one's computing career with Unix, say as an undergraduate, to being born in East Africa. It is intolerably hot, your body is covered with lice and flies, you are malnourished and you suffer from numerous curable diseases. But, as far as young East Africans can tell, this is simply the natural condition and they live within it. By the time they find out differently, it is too late. They already think that the writing of shell scripts is a natural act."

— Ken Pier, Xerox PARC

TECHNISCHE UNIVERSITÄT DRESDEN

Motivation (2)

“Two of the most famous products of Berkeley are LSD and Unix. I don't think this is a coincidence.”

-- Anonymous

- And we don't even talk about MS-DOS, Windows, BSD, ...
- OS decisions lead by requirements of 1960s and 1970s
- Singularity: Rethink systems design and interfaces, because hardware, software, computer users have evolved through the years.

TECHNISCHE UNIVERSITÄT DRESDEN

Singularity concepts

- Design strategies:
  - Safe programming language (Sing#)
  - Program verification
  - Improved system architecture

- Key concepts:
  - Software-isolated processes (SIP)
  - Contract-based channels
  - Manifest-based programs

TECHNISCHE UNIVERSITÄT DRESDEN

Software-isolated processes

- Like traditional OS processes:
  - execute code on behalf of a user
  - associated memory layout
  - may contain threads
  - information hiding, fault isolation
  - use Singularity System Call ABI
- Unlike traditional OS processes:
  - no shared (writable) memory
  - comm. via channels and exchange heap
  - code cannot be modified at exec. time
  - isolated by SW verification, no HW protection

TECHNISCHE UNIVERSITÄT DRESDEN

Contract-based channels

- bi-directional message channels with exactly two endpoints
- in-kernel, lossless, FIFO message queue
- protocols described by contracts (state machines) – compile-time verification
- no multiple sends allowed
- transmitted objects need to be allocated from special exchange heap – message is basically a change of object ownership (== Short IPC)

Manifest-based programs

- All code running in Singularity is described by a manifest.
  - description of code, system resources, capabilities, dependencies on other programs
- Installing code:
  - Check manifest if description matches system policies and common sense.
  - Compile application into native code (Bartok compiler)
  - Use signatures to detect modifications.
- Manifest also used to infer boot order of programs.

The Singularity kernel

- Microkernel approach: device drivers, network stacks, ... run outside the kernel.
- 192 system calls (explicit calls instead of multiplexing calls)
  - process creation
  - channel management
  - thread management
  - paging
  - synchronization
  - ...
- ABI versioning

The Singularity kernel (2)

- No traditional kernel entry/exit
  - privileged code can be “embedded” into SIPs at installation time
- Garbage collection
  - each SIP has a GC associated, can be chosen from a number of predefined GCs
  - Problem: OS and SIP may have different GCs, but share the same stack
    - delimiters on stack mark cross-AS calls, GC may simply skip over them

Compile-time Reflection

- Complete reflection has disadvantages:
  - runtime checking necessary
  - cannot optimize beforehand
  - can prevent system security policies by generating "evil" code at runtime
- CTR: transforms
  - templates -> well-formed, type-safe code
  - no need to use complex reflection APIs
- Use case: definition of device driver resource needs -> transformed into device driver startup code

Protection Domains

- Early versions of Singularity ran all code in Ring 0, no HW protection
- Now HW protection as well as SIPs can be used.

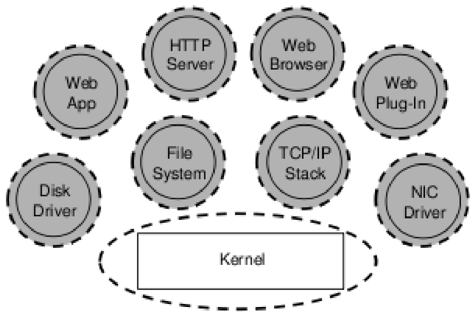

Figure 4a. Micro-kernel configuration (like MINIX 3). Dotted lines mark protection domains; dark domains are user-level, light are kernel-level.

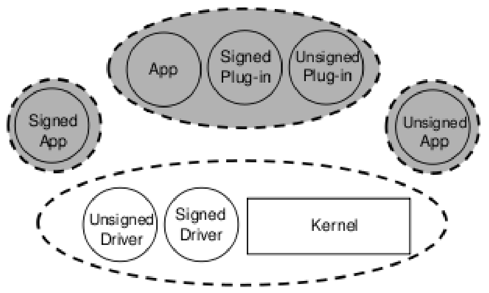

Figure 4b. Monolithic kernel and monolithic application configuration.

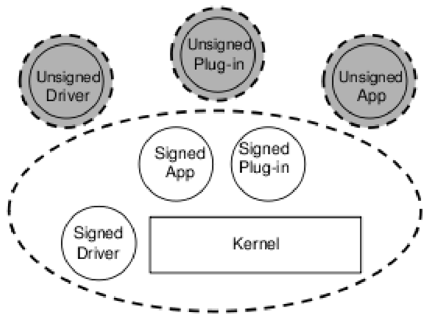

Figure 4c. Configuration with distinct policies for signed and unsigned code.

Rethink the Rethink...

- Is it really a microkernel?
- Fig. 5: Full microkernel has 37.7% overhead due to hardware protection domains & co.
  - Is this for all microkernels or only if you build a "real" microkernel using Singularity (e.g., no shared memory...)?
- Is tying SIPs to special processor cores the solution to all our problems?