Singularity

I. Background

New OS from scratch:
   o Not meant to be the next Vista, Windows 7, etc...
   o Intended as research vehicle for core OS ideas
   o Quite a large project over several years; still ongoing...
   o Primary goal: dependable computing

Three key ideas:
   o Depend heavily on type-safe memory-safe language and use it for isolation
   o Define contracts on channels between processes
   o Manifests define the exact set of code to run, with no new additions allowed

Kernel and all code implemented in Sing#, a variation of C#.
   o provides strong type and memory safety (similar to Java)
   o some extensions for contracts
   o Kernel ABI is trivial; all real services provided via channels

II. Software-Isolated Processes

Software-isolated processes (SIPs):
   o basic process isolation (but potentially in the same address space)
   o no shared memory among SIPs
   o communicated between SIPs using asynchronous message passing channels
   o system calls thus become async RPCs over a channel.
   o SIPs can be quite fine grained and low-cost to create (e.g. 50-byte object)

Hardware-based isolation is the traditional approach

Language-based safety:
   o Type safety: only allows the specific operations for a given instance of a type
   o Memory safety:
     • avoid NULL pointers
     • avoid array out-of-bounds
     • avoid references to deallocated memory (ie. no dangling refs)
Singularity

- all code must be verifiably safe
  - mostly ensured by the compiler/loader
  - some small amount of “untrusted” code in the kernel

Singularity design:
- keep SIPs independent
  - no shared memory
  - no shared pages
  - no shared allocator or garbage collector; each SIP could use a different GC mechanism
- can reclaim a SIP just be revoking its pages: all the objects for one SIP are entirely within its pages
- SIPs can be in the same address space and at the same privilege level (why is it OK to run the app with OS privileges?)
- Extensions and drivers run in their own SIP
  - 85% of Windows crashes due to non-isolated driver failures
- Could still use VM for demand paging
- Can also use hardware protection domains in addition to SIPs (belt & suspenders)

The performance costs of hardware-based isolation:
- General conclusion: more expensive than people think; with virtual memory and privilege levels costing more than just changing address spaces
- address translation: 10-30% (exception handling, TLB misses, kernel data structures, ...)
- changing address spaces? TLB misses, relatively small
- changing privilege levels? 4x a procedure call
- granularity? memory pressure
- clock speed? is TLB lookup on the critical path?
- Promotes shared-memory to avoid crossings: but shared memory reduces safety!
- IPC is very fast with SIPs, 30-500% for microbenchmarks, 20% for communication heavy benchmark
- Web server overall:
  - -6.3% for address translation,
  - -12.6% additionally for crossing protection boundary for the kernel,
  - -14% additionally for privilege levels,
  - for a total of 33% slower (37% if you use a microkernel approach)
- Overhead of safe language is about 4.6%

III. Channels

Communication is via message passing between SIPs (using a channel):
Singularity

- Semantically, this is pass by value
- Cannot pass an object reference; in fact can only pass structs or primitive types (not objects)
  - Two sides could in fact use different object layouts, different languages, etc.
- Construct a message in the “exchange heap” an area of memory intended for this purpose
- Exactly one pointer to a message (which gets passed to the receiver)
  - One pointer => sender can’t muck with message after it is sent
  - Based on linear types
  - Optimization: can pass by reference if in the same address space

Each endpoint of a channel is owned by one SIP (potentially the same SIP)
- Normally, both endpoints in the same address space, so pass by reference works
- If crossing a hardware boundary, then the data must be copied (sometimes twice: in and out of the kernel): 10-25 times higher than pass by ref

Channels are capabilities:
- You have to have the right channel endpoint to perform various tasks
- Example: opening a file returns a channel to access that file; there is no other way to access that file

Contracts: simple language extension to define a channel
- includes the message layout
- includes a finite-state machine for the sequence of legal states of the channel
- Each states lists what messages can arrive legally and what the resulting state will be (a direct encoding of an FSM)
- Compiler can statically verify that only the allowed messages are actually sent in each state; avoids run-time errors

IV. Manifests

Main idea: the code for an application is fixed at compile time -- no dynamic extensions and no dynamic code loading

This avoids the rampant problems with DLLs and versioned libraries. The manifest is a self-consistent invariant collection of code. To receive the value of an improve library, for example, someone must update the manifest; otherwise it will continue to use the old version.

Manifest captures:
- code
- required system resources
- desired capabilities
- dependencies on other programs
Singularity

Goal: don’t start a program that will need something it can’t get

Enables static verification:
  o example: new driver code is for a device that is available
  o names the channels that will be needed
  o verify the contract for those channels
  o verify no priviledges instructions, type safety, memory safety
  o correct version of kernel ABI

Code arrives as MSIL, which is a CPU-independent instruction set
  o after verification, loader compiles it to machine code