```
6.828 2014 Lecture 16: Singularity
==

Required reading:
    [Singularity](../readings/hunt07singularity.pdf)
        [Language support for message passing](../readings/singularity-eurosys2006.pdf)

Overview
--

Singularity is a Microsoft Research experimental O/S
  many people, many papers, reasonably high profile
  choice of problems maybe influenced by msft experience w/ windows
  we can speculate about influence on msft products

Stated goals
  increase robustness, security
    particularly w.r.t. extensions
  decrease unexpected interactions
  incorporate modern techniques

High level structure
  microkernel: kernel, processes, IPC
  they claim to have factored services into user processes (page 5)
    NIC, TCP/IP, FS, disk driver (sealing paper)
    kernel: processes, memory, some IPC, nameserver
    UNIX compatibility is not a goal, so avoiding some Mach pitfalls
  on the other hand there are 192 system calls (page 5)

Most radical part of design:
  Only one address space (paging turned off, no use of segments)
    kernel and all processes
  User processes run w/ full h/w privs (CPL=0)

Why is that useful?
  Performance
  Fast process switching: no page table switch
  Fast system calls: CALL not INT
  Fast IPC: no copying
  Direct user program access to h/w, for e.g. device drivers
  Table 1 shows they are a lot faster at microbenchmarks

But their main goal wasn't performance!
  robustness, security, interactions

Is *not* using pagetable protection consistent w/ goal of robustness?
  unreliability comes from *extensions*
    browser plug-ins, loadable kernel modules, &c
  typically loaded into host program's address space
    for speed and convenience
  so VM h/w already not relevant
  can we just do without hardware protection?

How would an extension work in Singularity?
  e.g. device driver, new network protocol, browser plug-in
  Separate process, communicate w/ host process via IPC

What do we think the challenges will be for single address space?
  Prevent evil or buggy programs from writing each other or kernel
  Support kill and exit -- avoid entangling

SIP
-
```

```
general SIP philosophy:
  "sealed"
  No modification from outside:
    none of JOS calls that take target envid argument (except start/stop)
    probably no debugger
    only IPC
  No modification from within:
    no JIT, no class loader, no dynamically loaded libraries

SIP rules
  only pointers to your own data
    no pointers to other SIP data or into kernel
    thus no sharing despite shared address space!
    limited exception for IPC messages in exchange heap
  SIP can allocate pages of memory from kernel
    different allocations are not contiguous

Why so crucial that SIPs can't be modified? Can't even modify themselves?
  What are the benefits?
    no code insertion attacks
    probably easier to reason about correctness
    probably easier to optimize, inline
      e.g. delete unused functions
    SIP can be a security principle, own files
  Is it worth the pain?

Why not like Java VM, can share all data?
  SIPs rule out all inter-process interactions
    except explicit via IPC
  SIPs more robust
  SIPs let every process have its own language run-time, GC scheme, &c
    though they are trusted and better not have bugs
    equivalent in sensitivity to kernel code
    so will be much harder for people to cook up their own
  SIPs make it easy for kernel to clean up after kill or exit

How to keep SIPs from reading/writing other SIPs?
  Only read/write memory the kernel has given you
  Have compiler generate code to check every access?
    "does this point to memory the kernel gave us?"
    Would slow code down (esp since mem isn't contig)
    We don't trust compiler

PL-based protection
---

the overall structure:
  1. compile to bytecodes
  2. verify bytecodes during install
  3. compile bytecodes -> machine code during install
  4. run the verified machine code w/ trusted runtime
  Why not compile to machine code?
  Why not JIT at run time?
  Why not verify at compile time?
  Why not verify at run time?

What does bytecode verification buy Singularity?
  Does it verify "only r/w memory kernel gave us"?
  Not exactly, but related:
    Only use reachable pointers [draw diagram]
    Cannot create a new pointer
      only trusted runtime can create pointers
    So if kernel/runtime never supply out-of-SIP pointers
      verified SIP can only use its own memory
  What does the verifier have to check to establish that?
```

```
    A. Don't cook up pointers (only use pointers someone gives you)
    B. Don't change mind about type
       Would allow violation of A, e.g. interpret int as pointer
    C. Don't use after free
       Re-use might change type, violate B
       Enforced with GC (and exchange heap linearity)
    D. Don't use uninitialized variables
    D. In general, don't trick the verifier
  Example?
      R0 <- new SomeClass;
      jmp L1
      ...
      R0 <- 1000
      jmp L1
      ...
    L1:
      mov (R0) -> R1
    potential problem:
      last mov is OK if via 1st jmp (assuming ptr legitimate)
        reads first element of SomeClass
      not OK if via 2nd jmp
        0x1000 may point into kernel
    verifier tries to deduce type for every register
      by pretending to execute along each code path
      requires that all paths to a reg use result in same type
      check that all reg uses OK for type
      would decide R0 has type int, or type SomeClass *
        either way, verifier would say "no"

Bytecode verification seems to do *more* than Singularity needs
  e.g. cooking up pointers might be OK, as long as within SIP's memory
  verifier may forbid some programs that might have been OK on Singularity
  Benefits of full verification:
    Fast execution, often don't need runtime checks at all
      Though some still needed: array bounds, OO casts, stack expansion
    Type check IPC types
    Need to allow r/w of exchange heap, but it is not SIP's memory
    Stack page allocation
    Do sys calls run on stack in SIP's memory?
      prevent thread X from wrecking thread Y's kernel syscall stack

You could put an interpreter in a SIP to evade ban on self-modifying code
  Would that cause trouble?

What parts are trusted vs untrusted?
  That is:
    All s/w has bugs
    Trusted s/w: if it has bugs, it can crash Singularity or wreck other SIPs
    Untrusted s/w: if it has bugs, can only wreck itself
  Let's consider some ordinary app, not a server.
    compiler. compiler output. verifier. verifier output. GC.

Exchange heap
--

Paper also talks about IPC
  How do SIPs communicate?
  endpoints, channels
  recv endpoint is a queue of messages
  message bodies are in "exchange heap"
  cool: no copy

Exchange heap is shared memory!
  What are the dangers?
  send the wrong type of data
```

```
  modify my msg to you while you are using it
  modify a totally unrelated message
  use up all exchange heap memory and don't free

How do they prevent abuse via exchange heap?
  verifier ensures SIP bytecodes keep only one ptr to anything in exchange heap
    never e.g. two
    and that SIP doesn't keep ptr after send()
      single-ptr rule helps here
    verifier knows when last ptr goes away
      via send
      via making another exchange heap obj point to it
      via delete
  single ptr rule prevents change-after-send
    and also ensures delete when done
  delete is explicit, no GC, but it's OK
    since verifier guarantees only a single ptr to each block
  runtime maintains owning-SIP entry in each exchg heap block
    updates on send() &c
    used to clean up in exit()

What are channel contracts for?
  Are they just nice to have, or do other parts of Singularity rely on them?
  The type signatures clearly are important.
    bytecode verifier (or something similar) must check them.
  The state machine part guarantees finite queues, no blocking send().
    and also catches protocol implementation errors
    e.g. sending msg when not expected

How does receive work?
  checks endpoints in shared mem, block on condition variable if no msgs
  so send must do a wakeup syscall

How do system calls into the kernel work?
  INT? CALL?
  what stack?
  since same stack, how does GC know?
  can a SIP pass pointers to kernel?

Endpoints function as capabilities
  Can pass them
  Can't talk to other SIPs w/o a channel
  Page 5 says they use channels to restrict access to e.g. files

Does evaluation support their claims?
  Robustness?
  Good model for extensions?
  Performance?
    e.g. real win from single address space, cheap syscall, switch, IPC
    Table 1, but only microbenchmarks
    Figure 5: unsafe code tax
      physical memory -- means paging disabled -- is this Singularity?
      Add 4KB pages -- means turn on paging, but single page table, all CPL=0
      separate domain -- separate page table for one of the SIPs, so switching costs
      ring 3 -- CPL=3 thus INT costs (for just one of the SIPs)
      full microkernel -- pgtable+INT for each of three SIPs
```