6.828 2012 Lecture 17: Language/OS Co-Design / Singularity

Why are we looking at this paper?
  completely different approach to isolation, protection
  language type-checking rather than hardware page protection

Singularity is a Microsoft Research experimental O/S
  many people, many papers, reasonably high profile
  stated goals:
    better robustness, security
    ground-up design w/ modern techniques

High level structure
  microkernel: SIP/thread mgmt, memory, IPC setup
    not so micro: 192 system calls (page 5)
  user-level service processes
    NIC, TCP/IP, FS, disk driver (sealing paper)
  not UNIX compatible

Most radical part of design:
  No hardware protection!
    Paging is turned off, so all memory visible to all instructions
    CPL=0, so can always run privileged instructions
  Instead: programming language protections

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

Q: why does no paging contribute to their main goal, robustness?

Remember what paging buys us:
  Protection
  Contiguous address space (starts at zero &c)
    Big arrays
    Contiguous stack
  Flexible address layout via mapping
  No fragmentation of physical memory
  Sharing/IPC via multiple mappings
  Tricks like copy-on-write fork, paging to disk

Challenges for no paging / CPL=0 design?
  Read/write only to appropriate memory
    And define what inappropriate means!
  Allow allocation and freeing of memory
  Allow interaction (IPC)
  But not too much entangling, so kill/exit can work

How to ensure SIP reads/writes only its own memory?

Q: why not have compiler generate check code before each load/store?
  speed, trust

The paper's approach:
 source
  compiler
 bytecodes
  install: verify and compile
 machine code
  trusted run-time
 running sip

Q: why not compile source to machine code?

Q: why verify/compile at install time? why not at run time -- JIT?

What properties does verification establish?
 Only use reachable pointers [draw diagram]
  only trusted runtime can create pointers
 So if kernel/runtime never supply out-of-SIP pointers
  verified SIP can only use its own memory

How does verification work?
 What does the verifier have to check?
  A. Don't invent or modify pointers
  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
  E. In general, don't trick the verifier
 Example bytecodes:
    R0 <- new SomeClass;
    jmp L1
    ...
    R0 <- 1000
    jmp L1
    ...
  L1:
    mov (R0) -> R1
  Q: is this code OK?
  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
  verifying the example:
    R0 has type SomeClass at first jmp to L1
    R0 has type integer at second jmp to L1
    so verifier would reject this code

Bytecode verification seems to do *more* than Singularity needs
 e.g. cooking up pointers might be OK, as long as within SIP's memory

so verifier may forbid some OK programs
this style of verification is off-the-shelf
enforcing exactly what Singularity needs is not
Singularity may actually need full verification
can't allow jump to data, even if data is in process's memory
since then executing unchecked code

What parts of verification scheme 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
Source?
Compiler?
Compiler output?
Verifier?
Machine code output of bytecode compiler?
Runtime / GC?

IPC: what would we want?
shared memory for efficiency
send complex data structures
but still have isolation, type checking

How do SIPs communicate?
IPC messages
"exchange heap" -- memory shared among all SIPs
thus zero-copy -- efficient
msgs can have pointers &c
thus can send complex data structures
each receiver has a queue in the exchange heap
send() system call to wake up receiving SIP
receiver blocks in recv() sys call, then checks queue

Q: dangers of shared-memory exchange heap?
write someone else's message
send the wrong type of data
modify my msg while you are reading it
use up all exchange heap memory and don't free

How do they prevent abuse via exchange heap?
verifier ensures SIP can only use ptrs someone gives it
i.e. only if you allocated mem, or found it in your recv queue
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()

Limitations of exchange heap idea
IPC can't carry existing language object -- not in exchange heap
Single-pointer rule limits the code you write
Need to use different types/functions for exchange heap data

What are channel contracts for? Section 2.2
Are they just nice to have, or do other parts of Singularity rely on them?
The type signatures clearly are important
they probably mesh with verified language types
perhaps you can't talk to a SIP that isn't verified to follow contract
The state machine part guarantees finite queues, no blocking send().
and also catches protocol implementation errors
e.g. sending msg when not expected

How do system calls into the kernel work?
INT? CALL?
what stack?
can a SIP pass pointers to kernel?
how does SIP GC know to not examine kernel part of stack?

Q: SIP allocates single pages -- how to have stack > 4096 bytes?

Q: How to have array > 4096 bytes?

2.1 says SIPs are "sealed"
Outlawed: JIT, dynamic library loader, self-modifying, debugger (?)
Q: Why is this important?
no code insertion attacks
maybe easier to reason about correctness
maybe easier to optimize, inline
e.g. delete unused functions
SIP can be a security principle, own files

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

Why not use a Java VM as your operating system?
Java has verification -- you can't make up pointers &c
Could have a Java thread for each running application

Singularity vs Java VM:
One SIP can *never* affect another SIP's memory
Not even with IPC
Would be easy to have such bugs w/ interacting Java threads
Exiting/killing a SIP releases all resources
Java must at least wait for a GC
SIPs let every process have its own language run-time, GC scheme, &c

What should the evaluation show?

What were their goals?
To gain robustness -- perhaps better than paging / CPL=3
To re-examine traditional design choices

What does the evaluation show?
Mostly about performance

Table 1 shows microbenchmark performance
10x reduction in sys calls -- why?
2x faster thread switch -- why?
5x faster IPC -- why?
2x-10x faster process creation -- why?

Figure 5: unsafe code tax
How much do they gain by static verification rather than run-time (or h/w) checks?
Simple file reading benchmark -- client SIP, file server SIP, device driver SIP
Figure 5 compare run-times; lower is better
physical memory -- Singularity (no paging, CPL=0, static verification, &c)
No runtime checks -- Singularity but no array bounds checking
Add 4KB pages -- paging enabled, 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
Figure 5 is useful: shows costs of various x86 features

What did we learn?
Are 1960s and 70s techniques now inadequate?
Should we use verification &c instead of paging hardware?
We *did* learn how to build O/S w/o paging -- very interesting!

A few open questions:
What are manifests for?
Can IPC carry a capability? How does kernel learn?
Does IPC receiver have to check msg format at run-time?
Why is the exchange heap data reference counted? When can the count be > 1?
When is it OK for SIP user code to execute a CPL=0 privileged instruction?

6.828 Operating System Engineering

Fall 2012