virtual machine (pt 2) / microkernels

last time (1)

sandboxing — filter system calls

guest OS running in hypervisor on host OS

hypervisor tracks virtual machine state

- does guest OS think it's in kernel mode?
- does guest OS think interrupts are enabled?
- ...

virtual machines: trap and emulate

- make some operation (IO, etc.) cause exception
- exception handler imitates operation
- e.g. read-from-keyboard-controller → host OS read() syscall
- e.g. system call → invoke guest OS syscall handler

last time (2)

virtual machine virtual memory

- virtual / physical / machine addresses

guest page table: virtual → physical

shadow page table: physical → machine

- possibly two: kernel/user

option one: fill shadow page table on demand

- guest OS indicates writes via TLB invalidations

option two: maintain shadow page table via trap-and-emulate

- mark guest page tables as read-only
- emulate write instruction to modify guest+shadow table

interlude: VM overhead

some things much more expensive in a VM:

I/O via priviliged instructions/memory mapping

- typical strategy: instruction emulation

exercise: overhead?

guest program makes read() system call

guest OS switches to another program

guest OS gets interrupt from keyboard

guest OS switches back to original program, returns from syscall

how many guest page table switches?

how many (real/shadow) page table switches?

hardware hypervisor support

Intel's VT-x

HW tracks whether a VM is running, how to run hypervisor

- new VMENTER instruction
- instruction switches page tables, sets program counter, etc.

HW tracks value of guest OS registers as if running normally

new VMEXIT interrupt — run hypervisor when VM needs to stop

- exits 'VM is running mode', switch to hypervisor

hardware hypervsior support

VMEXIT triggered regardless of user/kernel mode

- means guest OS kernel mode can't do some things
- real I/O device, unhandled priviliged instruction, ...

partially configurable: what instructions cause VMEXIT

- reading page table base? writing page table base? ...

partially configurable: what exceptions cause VMEXIT

- otherwise: HW handles running guest OS exception handler instead

no VMEXIT triggered? guest OS runs normally (in kernel mode!)

HW help for VM page tables

already avoided two shadow page tables:

- HW user/kernel mode now separate from hypervisor/guest

but HW can help a lot more

tagged TLBs

hardware includes "address space ID" in TLB entries

also helpful for normal OSes — faster context switching

hypervisor and/or OS sets address space ID when switching page tables

extra work for OS/hypervisor:

- need to flush TLB entries even when changing non-active page tables

nested page tables

virtual → physical → machine

hypervisor specifies two page table base registers

- guest page table base — as physical address
- hypervisor page table base — as machine address

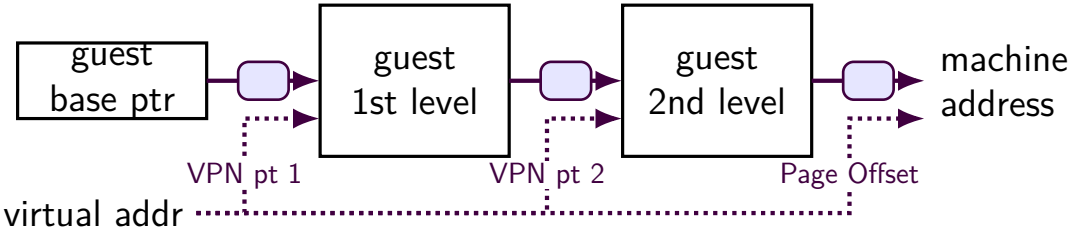
guest page table contains physical (not machine) addresses

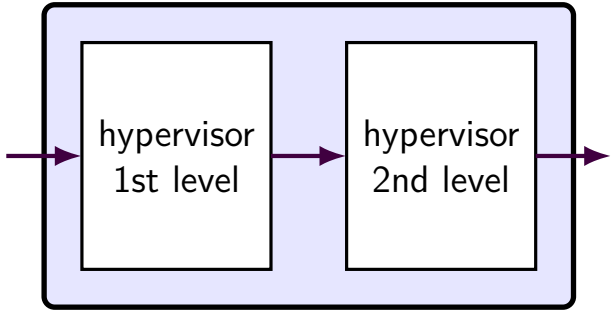
hardware walks guest page table using hypervisor page table

- guest page table contains physical addresses
- hardware translates each physical page number to machine page number

nested 2-level page tables: how many lookups?

nested 2-level tables





non-virtualization instrs.

assumption: priviliged operations cause exception instead

and can keep memory mapped I/O to cause exception instead

many instructions sets work this way

x86 is not one of them

POPF

POPF instruction: pop flags from stack

- condition codes — CF, ZF, PF, SF, OF, etc.
- direction flag (DF) — used by "string" instructions
- I/O privilege level (IOPL)
- interrupt enable flag (IF)
- ...

POPF

POPF instruction: pop flags from stack

- condition codes — CF, ZF, PF, SF, OF, etc.
- direction flag (DF) — used by "string" instructions
- **I/O privilege level** (IOPL)
- **interrupt enable flag** (IF)
- ...

some flags are privileged!

popf **silently** doesn't change them in user mode

PUSHF

PUSHF: push flags to stack

write actual flags, include privileged flags

hypervisor wants to pretend those have different values

handling non-virtualizable

option 1: patch the OS

- typically: use hypervisor syscall for changing/reading the special flags, etc.
- 'paravirtualization'
- minimal changes are typically very small — small parts of kernel only

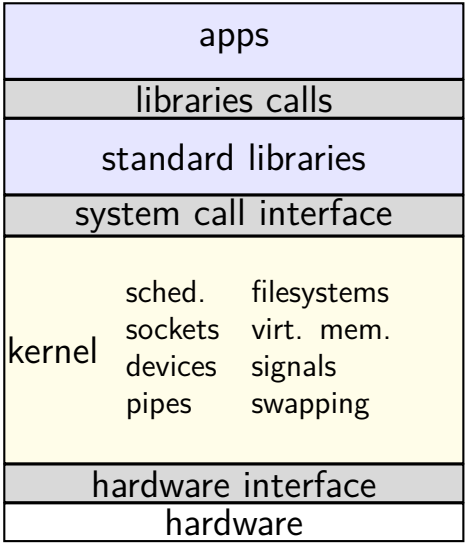
option 2: binary translation

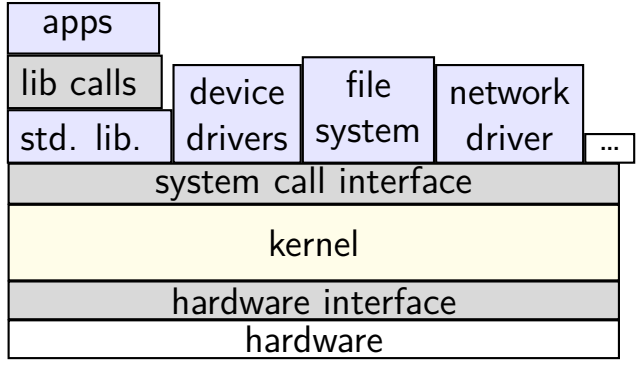
- compile machine code into new machine code

option 3: change the instruction set

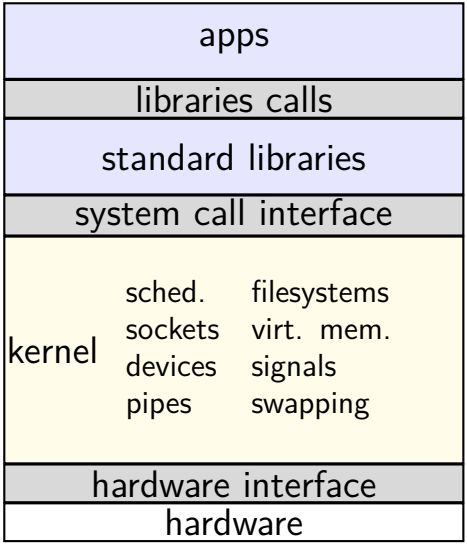
- after VMs popular, extensions made to x86 ISA
- one thing extensions do: allow changing how push/popf behave

monolithic versus microkernel

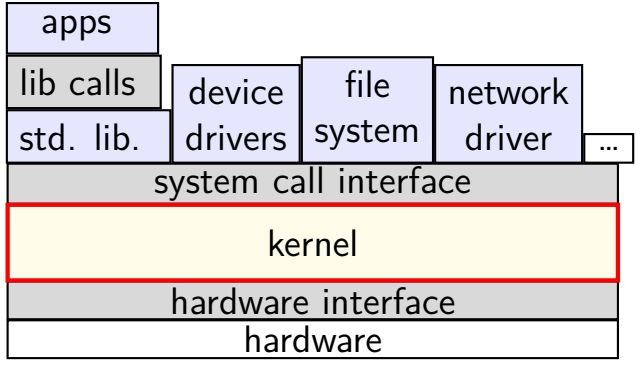




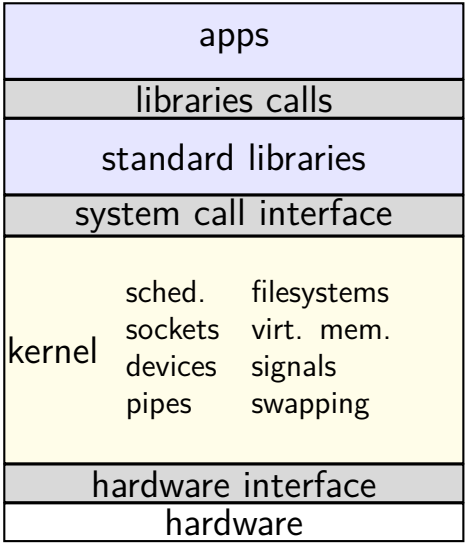
monolithic versus microkernel



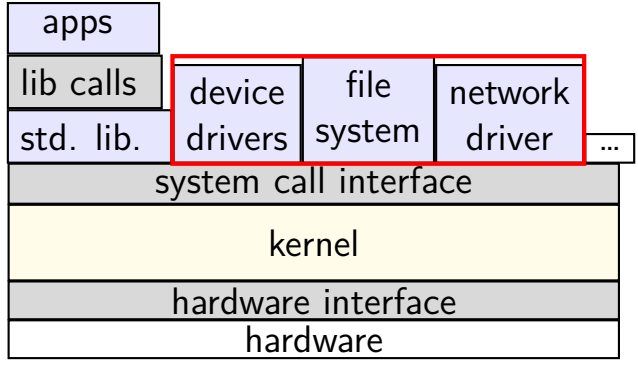
microkernel
minimal functionality in kernel mode



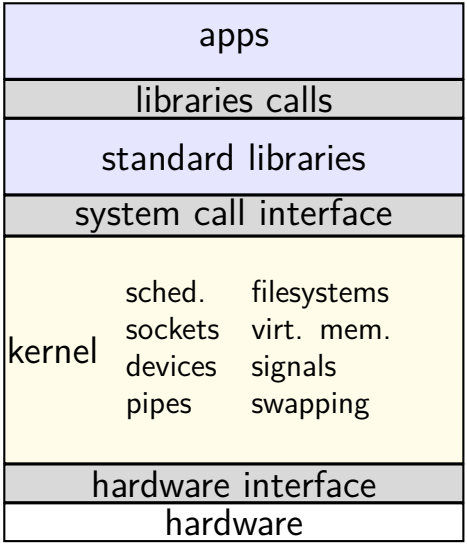
monolithic versus microkernel



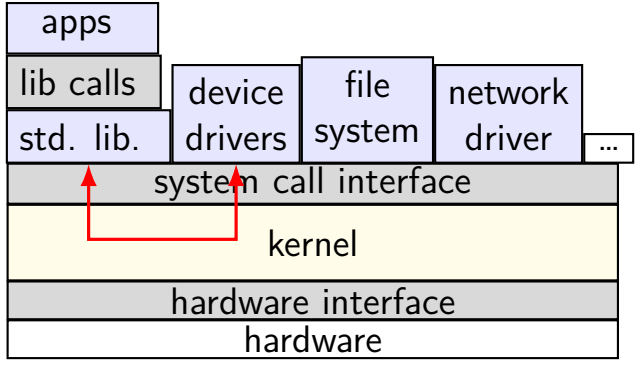
device drivers are separate proceses run in userspace? more modular?



monolithic versus microkernel



kernel provides fast communication to device drivers, etc.



microkernel services

interprocess communication
- performance is very important
- used to communicate with *OS services*

raw access to devices
- map device controller memory to device drivers
- forward interrupts

CPU scheduling
- tied to interprocess communication

virtual memory

hope: everything else handled by userspace servers

microkernel services

physical memory access

- including device controller acccess

CPU scheduling

interrupts/exceptions access

communication

synchronization

seL4

example microkernel: seL4

notable as *formally verified*

- machine-checked proof of some properties

uses microkernel design

seL4 system calls (full list)

send message: Send, NBSend, Reply

recv message: Recv, NBRecv

send+recv message: Call, ReplyRecv

to avoid requiring two syscalls

Yield() (run scheduler)

seL4 kernel services?

but how to allocate memory, threads, etc.???

can send messages to *kernel objects*

- same syscall as talking to device driver, other app, etc.

seL4 naming

where to send/recv from?

seL4 answer: capabilities

- opaque tokens $\sim$ file descriptors
- indicate allowed operations (read, write, etc.)

represent everything

- other processes
- kernel objects (= thread, physical memory, ...)

can be passed in messages

seL4 naming

where to send/recv from?

seL4 answer: capabilities

- opaque tokens $\sim$ file descriptors
- indicate allowed operations (read, write, etc.)

represent everything

- other processes
- kernel objects (= thread, physical memory, ...)

can be passed in messages

seL4 objects

kernel objects — named via capability

have “methods”

- invoked via Sending message

seL4 kernel objects (x86-3)

capability storage — `Cnode`

threads — `TCB` (thread control block)

IPC — `Endpoint`, `Notification`

virtual memory — `PageDirectory`, `PageTable`

available memory — `Frame`, `Untyped`

interrupts — `IRQControl`, `IRQHandler`

(and a few more)

seL4 choices

abstract hardware pretty directly

- expose page table structure, interrupts, etc.
- let libraries, userspace services handle making interface generic

no kernel memory allocation

- userspace code controls how physical memory is assigned
- ...including memory for kernel objects!

seL4 choices

abstract hardware pretty directly

- expose page table structure, interrupts, etc.
- let libraries, userspace services handle making interface generic

no kernel memory allocation

- userspace code controls how physical memory is assigned
- ...including memory for kernel objects!

seL4 object conversion

most memory starts as `Untyped` objects

cannot, e.g., just say "make a new TCB"

instead: derive TCB from `Untyped`

= allocate TCB in this memory

cannot say "allocate me memory"

instead: derive `Frame` from `Untyped`

= allocate Frame (physical page) in this memory

...and add `Frame` to `PageTable`

seL4 capabilities

objects represented by capabilities

capability takes slot in Cnode

- capability storage — like file descriptor table

can copy capabilities

- and drop some permissions (e.g. read-only copy)

can copy derived capabilities to other processes

seL4 object deletion?

what about deleting objects

- capability $\approx$ pointer to object

kernel tracks reference count of every object

reference count $= 0 \rightarrow$ original deleted

- available again via Untyped object

seL4 object deletion?

what about deleting objects

- capability $\approx$ pointer to object

kernel tracks reference count of every object

reference count $= 0 \rightarrow$ original deleted

- available again via Untyped object

deleting Cnode (capability table)? recursive deletion

derived capabilities and revocation

kernel tracks "children" of capabilities

example: endpoint of device driver copied to many clients

revoking parent capability

also deletes all children

- try to access server now? "sorry, it's closed"

derived capabilities (figure)

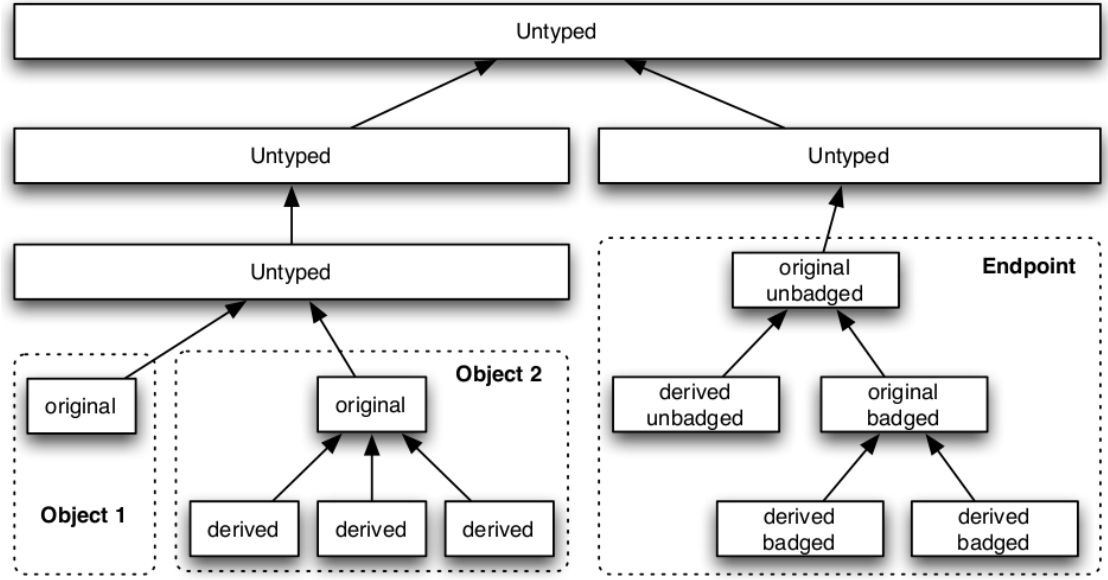


figure from seL4 manual

seL4 messages

"tag" — message type + size

"badges" — identifying source

- multiple virtual endpoints which go to same server
- badge says which sender used

one or more "message words"

- first few stored in CPU registers (for speed)
- additional ones stored in per-thread buffer

one or more capabilities

seL4 IPC destinations

each kernel object is message destination

- method invocation = send message + recieve reply
- can imitate kernel object perfectly with user server

server endpoints are *badged*

- server gives out different badge for each client
- allows one server to handle multiple services
- way to add badges when handing out capabilities

seL4 IPC destinations

each kernel object is message destination

- method invocation = send message + recieve reply
- can imitate kernel object perfectly with user server

server endpoints are *badged*

- server gives out different badge for each client
- allows one server to handle multiple services
- way to add badges when handing out capabilities

synchronous IPC

seL4 messages are *synchronous*

Send() waits for corresponding Recv() to happen

advantage: message not copied into kernel buffer

advantage: handle message by context switching to target process

synchronous IPC

seL4 messages are *synchronous*

Send() waits for corresponding Recv() to happen

advantage: message not copied into kernel buffer

advantage: handle message by context switching to target process

fast path: message entirely in all in registers

fast path: scheduler switches directly from sender to receiver

Send() cases

Send() to kernel object: invoke kernel handler, reply

Send() to program ready to recieve: just context switch now

Send() to program not ready to recieve: add thread to queue
then context switch to something else, Send() always blocks

Send() to invalid destination: reply with error

SendRecv() optimization

system call combining Send() + Recv()

ideal usage:

context switch to service to Send()

service handles message and replies

context switch from service to Recv()

combined system call: ready to recieve immediately after Send()

- always using "just context switch now" code

notifications: async IPC

seL4 message passing is synchronous

seL4 also supports simple asynchronous IPC

Notification = bianry semaphores

Signal (up) and Wait (down) operations

special: can signal/wait on multiple semaphores at once

- e.g. wait for one of several events

notifications versus messages

notifications don't block

signal and forget

- not possible for message Send()!

multiple threads can wait at once

- possibly easier than messages for coordinating?

seL4 virtual memory: do it yourself

Thread associated with PageDirectory+PageTable objects

send messages to object to map pages

kernel tracks reference counts

- can share pages between threads

sel4 virtual memory: page faults?

what about copy-on-write?

you can do that yourself!

sel4 virtual memory: page faults?

what about copy-on-write?

you can do that yourself!

each thread as exception endpoint

exceptions become message-sends

- can setup page-fault-handler thread/server
- give it capabilities to your PageDirectory

userspace page fault handlers

message sent to page-fault handler thread

page fault at address X accessing address Y ...

thread uses PageDirectory/PageTable objects

then replies to message — restarting original thread

userspace page fault handlers

message sent to page-fault handler thread

- page fault at address X accessing address Y ...

thread uses PageDirectory/PageTable objects

then replies to message — restarting original thread

same applies to other exceptions

- divide by zero, illegal instruction, etc.

seL4 IO and Interrupts

I/O: give device drivers Frames for device controller memory

kernel forwards interrupts as messages

provides protocol for acknowledging interrupts

(poorly?) selected other OS designs

Exokernel (late 90s)

- kernel's only job is sharing hardware
- *no attempt* to abstract hardware resources
- explicit resource revocatoin

Singularity

- OS is a language virtual machine interpreter
- no virtual memory

something for datacenters/manycore?

Exokernel

heavily certainly influenced seL4's design

key idea: kernel only securely multiplexes (shares) resources

programs have a "library operating system" to talk to kernel

Exokernel philosophy

kernel provides almost exactly the hardware interface

- direct access if safe

kernel's only job: filter hardware usage for safety

- safety: your program doesn't access things it shouldn't

program libraries handle all abstractions

Exokernel: memory multiplexing

capabilities for physical memory pages (like seL4)

use capability to request virtual to physical mappings (like seL4)

but...kernel can *take back* pages

tells library operating system "I'm going to need a page back"

library operating system needs to deallocate a page

- if it doesn't quickly enough — reclaim by force (lost data?)

Exokernel: network multiplexing

kernel doesn't implement sockets — only raw "send message"

kernel filters outgoing packets sent by programs

filter = port numbers you are assigned

library operating system handles all details of sockets

Exokernel: CPU multiplexing

kernel does not keep thread control blocks

instead: library OS says "start running here"

- library OS has its own "start the right thread" code

library OS supplies 'exception' handling code locations

e.g., on timer expiration:

- kernel runs library OS "stop running now" handler
- if that code doesn't yield to OS quickly, then kernel kills program

e.g., on IO event:

- kernel runs library OS "I/O event happened" handler
- library OS can do context switch itself

Singularity

Microsoft Research (2003-2010)

OS runs CIL (Common Intermediate Language) code

- bytecode, similar idea to Java bytecode

software-based isolation

- no page tables at all
- rely on bytecode to keep processes from access each other's memory

probably has huge issues with recently discovered Spectre/etc. attacks

Singularity: performance arguments

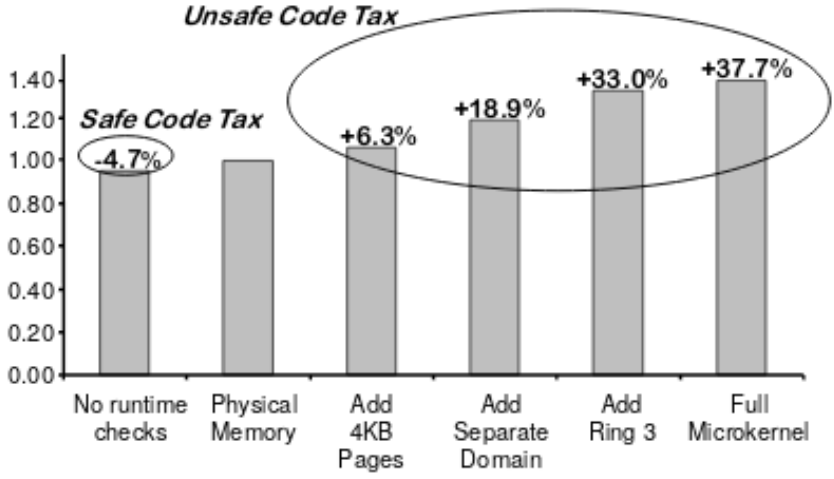


Figure 5. Normalized WebFiles execution time.

from Hunt and Larus, "Singularity: Rethinking the Software Stack" (2006)

Singularity issues

is software-based isolation trustable?

need to verify bytecode $\rightarrow$ machine code compiler

- but only enough to prove memory-safety/etc.

Specture/Meltdown = information leaks through caches, etc.

- probably need hardware isolation to prevent these
- not known when Singularity prototyped

datacenter OS ideas?

OS distributed across multiple servers?

- especially attractive with very fast interconnections (e.g. PCI)

OSes specialized for running virtual machines?

manycore OS ideas?

future of thousands of cores?

want to schedule *many cores together*

- hypothesis: efficient applications use multiple cores at once

faster to talk to other core than context switch?

- store+load into shared cache versus context switch

backup slides

binary translation

compile assembly to new assembly

works without instruction set support

early versions of VMWare on x86

later, x86 added HW support for virtualization

multiple ways to implement, I'll show one idea

- similar to Ford and Cox, "Vx32: Lightweight, User-level Sandboxing on the x86"

binary translation idea

```
0x40FE00: addq %rax, %rbx
movq 14(%r14,4), %rdx
addss %xmm0, (%rdx)
...
0x40FE3A: jne 0x40F404
subss %xmm0, 4(%rdx)
...
je 0x40F543
ret
```

binary translation idea

```
0x40FE00: addq %rax, %rbx
movq 14(%r14,4), %rdx
addss %xmm0, (%rdx)
...
0x40FE3A: jne 0x40F404
```

```
subss %xmm0, 4(%rdx)
...
je 0x40F543
```

```
ret
```

divide machine code
into *basic blocks*
(= "straight-line" code)
(= code till
jump/call/etc.)

binary translation idea

```
0x40FE00: addq %rax, %rbx
movq 14(%r14,4), %rdx
addss %xmm0, (%rdx)
...
0x40FE3A: jne 0x40F404
subss %xmm0, 4(%rdx)
...
je 0x40F543
ret
```

generated code:

```
// addq %rax, %rbx
movq rax_location, %rdi
movq rbx_location, %rsi
call checked_addq
movq %rax, rax_location
...
// jne 0x40F404
... // get CCs
je do_jne
movq $0x40FE3F, %rdi
jmp translate_and_run
do_jne:
movq $0x40F404, %rdi
jmp translate_and_run
```

a binary translation idea

convert whole *basic blocks*

- code upto branch/jump/call

end with call to `translate_and_run`

- compute new simulated PC address to pass to call

making binary translation fast

only have to convert kernel code

- and only some of the kernel code

cache converted code

- `translate_and_run` checks cache first

patch calls to `translate_and_run` to jmp to cached code

do something more clever than `movq rax_location, ...`

- map (some) registers to registers, not memory

ends up being "just-in-time" compiler