x86 ASM (1)

last time: VMs

configurablity

consistent environment

isolation

run malware in safe environment

key mechanism: hardware support

hardware gives VM monitor control when needed

backup mechanism: emulation, binary translation

from last time: POPF

on POPF

user mode: silently doesn't change interrupt enable flag (`IF`)

(and some other "privileged" flags)

kernel mode: does change `IF`

what VM wants:

call VM monitor to change simulated `IF`

from last time: the VM stack

guest OS machine code runs on real hardware

"normal" instructions:

- run directly on real hardware

privileged operations:

- real hardware gives control to host OS
- host OS gives control to VM monitor

from last time: binary translation

key idea: machine code to new machine code

- change/translate as needed

in small segments: end by returning to translator

- avoid needing to translate whole program
- handle things like loading new code

efficient version:

- cache translated code
- replace calls to translator with jumps to translated code

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
```

x86-64 assembly

your assignments will use it

will have examples with lots of assembly

x86-64 machine code will come up

x86-64 assembly

history: AMD constructed 64-bit extension to x86 first

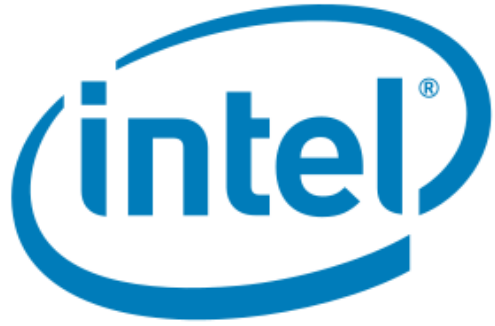
- marketing term: AMD64

Intel first tried a new ISA (Itanium), which failed

Then Intel copied AMD64

- marketing term: EM64T
  - Extended Memory 64 Technology
- later marketing term: Intel 64

both Intel and AMD have manuals — definitive reference



Intel® 64 and IA-32 Architectures Software Developer's Manual

Combined Volumes:
1, 2A, 2B, 2C, 2D, 3A, 3B, 3C and 3D

x86-64 manuals

Intel manuals:

`https://software.intel.com/en-us/articles/intel-sdm`

24 MB, 4684 pages

Volume 2: instruction set reference (2190 pages)

AMD manuals:

`https://support.amd.com/en-us/search/tech-docs`

"AMD64 Architecture Programmer's Manual"

example manual page

INC—Increment by 1

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FE /0	INC *r/m8*	M	Valid	Valid	Increment *r/m* byte by 1.
REX + FE /0	INC *r/m8**	M	Valid	N.E.	Increment *r/m* byte by 1.
FF /0	INC *r/m16*	M	Valid	Valid	Increment *r/m* word by 1.
FF /0	INC *r/m32*	M	Valid	Valid	Increment *r/m* doubleword by 1.
REX.W + FF /0	INC *r/m64*	M	Valid	N.E.	Increment *r/m* quadword by 1.
40+ *rw***	INC *r16*	O	N.E.	Valid	Increment word register by 1.
40+ *rd*	INC *r32*	O	N.E.	Valid	Increment doubleword register by 1.

NOTES:

* In 64-bit mode, r/m8 can not be encoded to access the following byte registers if a REX prefix is used: AH, BH, CH, DH.

** 40H through 47H are REX prefixes in 64-bit mode.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
M	ModRM:r/m (r, w)	NA	NA	NA
O	opcode + rd (r, w)	NA	NA	NA

Description

Add 1 to the destination operand, while preserving the state of the CF flag. The destination operand can be a

instruction listing parts (1)

opcode — first part of instruction encoding

- yes, variable length
- "REX"???
- more later (today or next week)

instruction — Intel assembly skeleton

`r/m32` — 32-bit memory or register value

64-bit mode — does instruction exist in 64-bit mode?

compat/leg mode — in 16-bit/32-bit modes?

instruction listing parts (2)

dscription + operation (later on page)

- text and pseudocode description

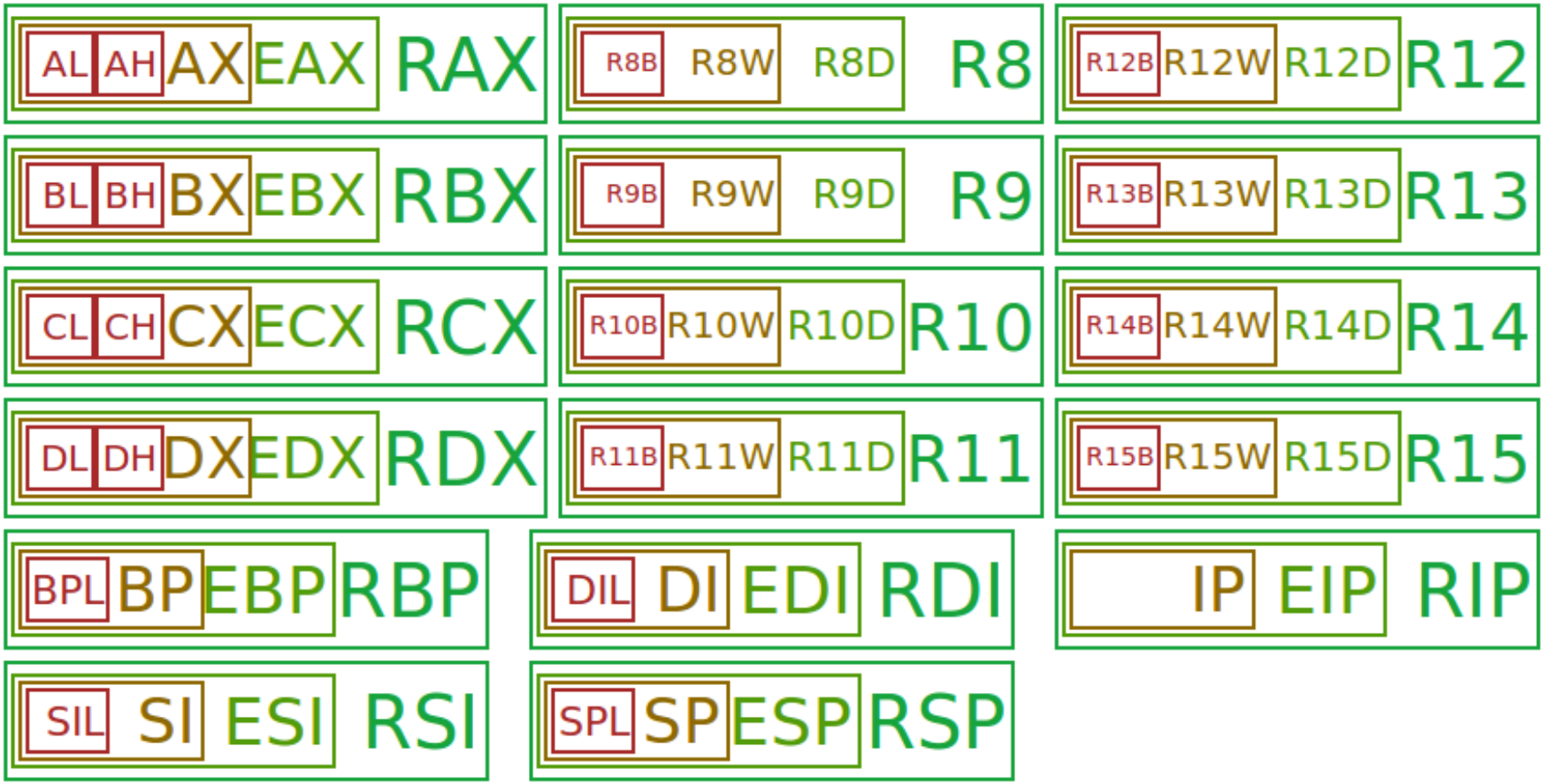
flags affected

- flags — used by `jne`, etc.

exceptions — how can OS be called from this?

- example: can invalid memory access happen?

recall: x86-64 general purpose registers



Immae via Wikipedia

overlapping registers (1)

setting 32-bit registers sets whole 64-bit register

extra bits are always zeroes

```
movq $0x123456789abcdef, %rax
xor %eax, %eax
// %rax is 0, not 0x1234567800000000
movl $-1, %ebx
// %rbx is 0xFFFFFFFF, not -1 (0xFFFFF...FFF)
```

overlapping registers (2)

setting 8/16-bit registers doesn't change rest of 64-bit register:

```
movq $0x12345789abcdef, %rax
movw $0xaaaa, %ax
// %rax is 0x123456789abaaaa
```

AT&T versus Intel syntax

AT&T syntax:

```
movq $42, 100(%rbx,%rcx,4)
```

Intel syntax:

```
mov QWORD PTR [rbx+rcx*4+100], 42
```

effect (pseudo-C):

```
memory[rbx + rcx * 4 + 100] <- 42
```

AT&T syntax (1)

```
movq $42, 100(%rbx,%rcx,4)
```

destination last

constants start with $

registers start with %

AT&T syntax (2)

```
movq $42, 100(%rbx,%rcx,4)
```

operand length: `q`

`l` = 4; `w` = 2; `b` = 1

`100(%rbx,%rcx,4)`:
`memory[100 + rbx + rcx * 4]`

`sub %rax, %rbx`: `rbx` ← `rbx - rax`

Intel syntax

destination first

`[...]` indicates location in memory

`QWORD PTR [...]` for 8 bytes in memory

- DWORD for 4
- WORD for 2
- BYTE for 1

On LEA

LEA = Load Effective Address

uses the syntax of a memory access, but...

just computes the address and uses it:

`leaq 4(%rax), %rax` same as
`addq` `$4, %rax`

almost — doesn't set condition codes

LEA tricks

`leaq (%rax,%rax,4), %rax` multiplies `%rax` by 5

```
address-of(memory[rax + rax * 4])
```

`leal (%rbx,%rcx), %eax` adds rbx + rcx into eax

ignores top 64-bits

question

```
.data
string:
    .asciz "abcdefgh"
.text
    movq $string, %rax
    movq string, %rdx
    movb (%rax), %bl
    leal 1(%rbx), %ebx
    movb %bl, (%rax)
    movq %rdx, 4(%rax)
```

What is the final value of string?

a. "abcdabcd"
b. "bbcdefgh"
c. "bbcdabcd"
d. "abcdefgh"
e. something else / not enough info

Linux x86-64 calling convention

System V Application Binary Interface

AMD64 Architecture Processor Supplement

Draft Version 0.99.7

Edited by

Michael Matz[1], Jan Hubička[2], Andreas Jaeger[3], Mark Mitchell[4]

November 17, 2014

Linux x86-64 calling summary

first 6 arguments: %rdi, %rsi, %rdx, %rcx, %r8, %r9

additional arguments: push on stack

return address: push on stack

`call`, `ret` instructions assume this

return value: %rax

caller-saved registers

functions may freely trash these

return value register `%rax`

argument registers:

`%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`

`%r11`

MMX/SSE/AVX registers: `%xmm0-15`, etc.

floating point stack: `%st(0)-%st(7)`

condition codes (used by `jne`, etc.)

callee-saved registers

functions must preserve these

`%rsp` (stack pointer), `%rbp` (frame pointer, maybe)

`%r12`–`%r15`

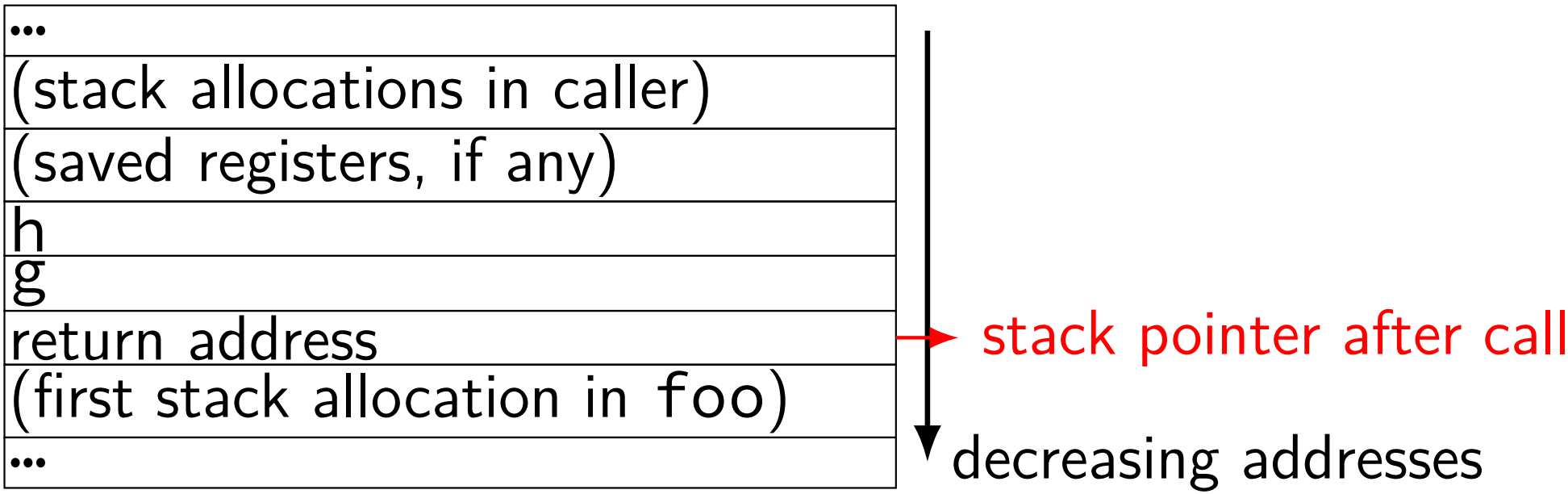
caller/callee-saved

```
foo:
    pushq %r12 // r12 is caller-saved
    ... use r12 ...
    popq %r12
    ret


...
other_function:
    pushq %r11 // r11 is caller-saved
    ...
    callq foo
    popq %r11
```

the call stack

```
foo(a,b,c,d,e,f,g,h);
```



calling convention example

```
int foo(int a, int b, int c, int d, int e, int f,
...
foo(1, 2, 3, 4, 5, 6, 7, 8);

pushq   $8
pushq   $7
movl    $6, %r9d
movl    $5, %r8d
movl    $4, %ecx
movl    $3, %edx
movl    $2, %esi
movl    $1, %edi
call    foo
/* return value in %eax */
```

floating point operations

x86 has two ways to do floating point

method one — legacy: x87 floating point instructions

- still common in 32-bit x86

method two — SSE instructions

- work more like what you expect

x87 floating point stack

x87: 8 floating point registers
`%st(0)` through `%st(7)`

arranged as a stack of registers

example: **`fld`** `0(%rbx)`

	before	after
`st(0)`	5.0	(value from memory at `%rbx`)
`st(1)`	6.0	5.0
`st(1)`	7.0	6.0
…	…	…
`st(6)`	10.0	9.0
`st(7)`	11.0	10.0

x87

not going to talk about x87 more in this course

essentially obsolete with 64-bit x86

SSE registers

SSE and SSE2 extensions brought vector instructions

```
numbers: .float 1 .float 2 .float 3. float 4
ones:    .float 1 .float 3 .float 5 .float 7
result:  .float 0 .float 0 .float 0 .float 0
...
movps numbers, %xmm0
movps ones, %xmm1
addps %xmm1, %xmm0
movps %xmm0, result
/* result contains: 1+1=2,2+3=5,3+5=8,4+7=11 */
```

SSE registers

SSE and SSE2 extensions brought vector instructions

```
numbers: .float 1 .float 2 .float 3. float 4
ones:    .float 1 .float 3 .float 5 .float 7
result:  .float 0 .float 0 .float 0 .float 0
...
movps numbers, %xmm0
movps ones, %xmm1
addps %xmm1, %xmm0
movps %xmm0, result
/* result contains: 1+1=2,2+3=5,3+5=8,4+7=11 */
```

array of 4 floats

SSE registers

SSE and SSE2 extensions brought vector instructions

```
numbers: .float 1 .float 2 .float 3. float 4
ones:    .float 1 .float 3 .float 5 .float 7
result:  .float 0 .float 0 .float 0 .float 0
...
movps numbers, %xmm0
movps ones, %xmm1
addps %xmm1, %xmm0
movps %xmm0, result
/* result contains: 1+1=2,2+3=5,3+5=8,4+7=11 */
```

move packed single
(single-precision float)

SSE registers

SSE and SSE2 extensions brought vector instructions

```
numbers: .float 1 .float 2 .float 3. float 4
ones:    .float 1 .float 3 .float 5 .float 7
result:  .float 0 .float 0 .float 0 .float 0
...
movps numbers, %xmm0
movps ones, %xmm1
addps %xmm1, %xmm0
movps %xmm0, result
/* result contains: 1+1=2,2+3=5,3+5=8,4+7=11 */
```

add packed single
(single-precision float)

XMM registers

`%xmm0` through `%xmm15` (`%xmm8` on 32-bit)

each holds 128-bits —

- 32-bit floating point values (`addps`)
- 64-bit floating point values (`addpd`)
- 64/32/16/8-bit integers (`paddq`/`d`/`w`/`b`)
- a 32-bit floating point value, 96 unused bits (`addss`)
- a 64-bit floating point value, 64 unused bits (`addsd`)

XMM registers

`%xmm0` through `%xmm15` (`%xmm8` on 32-bit)

each holds 128-bits —

- 32-bit floating point values (`addps`)
- 64-bit floating point values (`addpd`)
- 64/32/16/8-bit integers (`paddq`/`d`/`w`/`b`)
- a 32-bit floating point value, 96 unused bits (`addss`)
- a 64-bit floating point value, 64 unused bits (`addsd`)

FP example

```
multiplyEachElementOfArray:
/* %rsi = array, %rdi length,
   %xmm0 multiplier */
loop:   test %rdi, %rdi
        je done
        movss (%rsi), %xmm1
        mulss %xmm0, %xmm1
        movss %xmm1, (%rsi)
        subq $1, %rdi
        addq $4, %rsi
        jmp loop
done:   ret
```

floating point calling convention

use `%xmm` registers in order

note: variadic functions

variable number of arguments

- `printf`, `scanf`, ...
- see `man stdarg`

same as usual

...but `%rax` contains number of `%xmm` used

AVX

recent Intel/AMD processors implement the AVX extension

adds `%ymm` registers

256-bit versions of `%xmm` registers

XMM0 is a name for the bottom 128 bits of YMM0

like RAX and EAX, or RDX and EDX

(later extension: even larger zmm registers)

addressing modes (1)

`$constant`

`displacement(%base, %index, scale)`

- `displacement` (absolute)
- `displacement(%base)`
- `displacement(,%index, scale)`

$(= \text{displacement} + \text{base} + \text{index} \times \text{scale})$

addressing modes (2)

`displacement(%rip)` (64-bit only)

```
thing: .quad 42
...
movq thing(%rip), %rax
```

encoded as offset from address of next instruction
(normally: label encoded as 32 or 64-bit address)
helps relocatable code

addressing modes (3)

`jmp *%rax` (or `call`)

- Intel syntax: `jmp RAX`
- what you'd expect to be `jmp (%rax)`

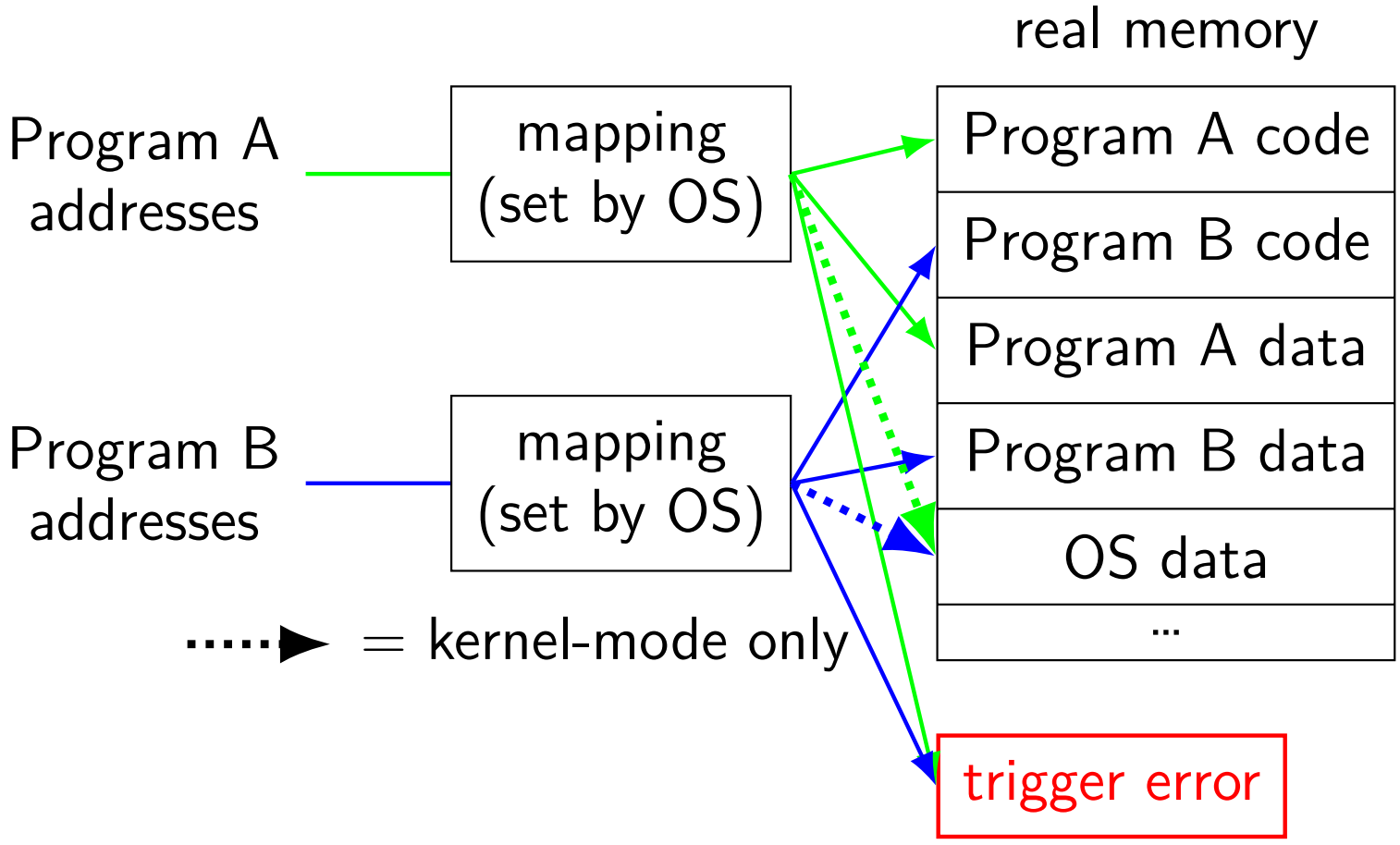
`jmp *(%rax)`

- read value from memory at RAX
- PC becomes location in that value
- Intel syntax: `jmp [RAX]`

`jmp *(%rax,%rbx,8)`

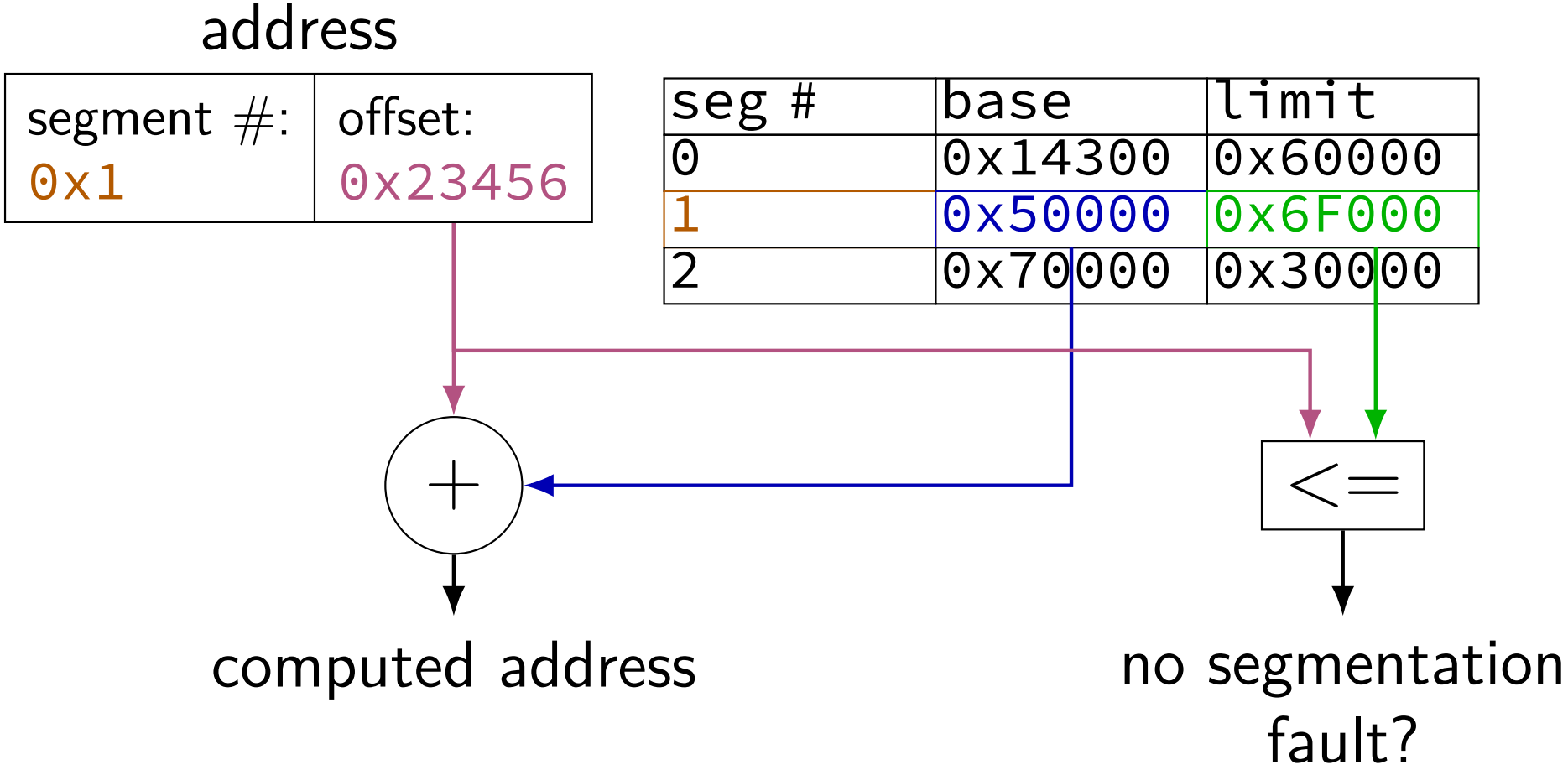
recall(?): virtual memory

illuision of dedicated memory



segmentation

before virtual memory, there was segmentation



x86 segmentation

addresses you've seen are the offsets

but every access uses a segment number!

segment numbers come from registers

- CS — code segment number (jump, call, etc.)
- SS — stack segment number (push, pop, etc.)
- DS — data segment number (mov, add, etc.)
- ES, FS, GS — extra segments (never default)

instructions can have a segment override:

```
movq $42, %fs:100(%rsi)
    // move 42 to segment (# in FS),
    // offset 100 + RSI
```

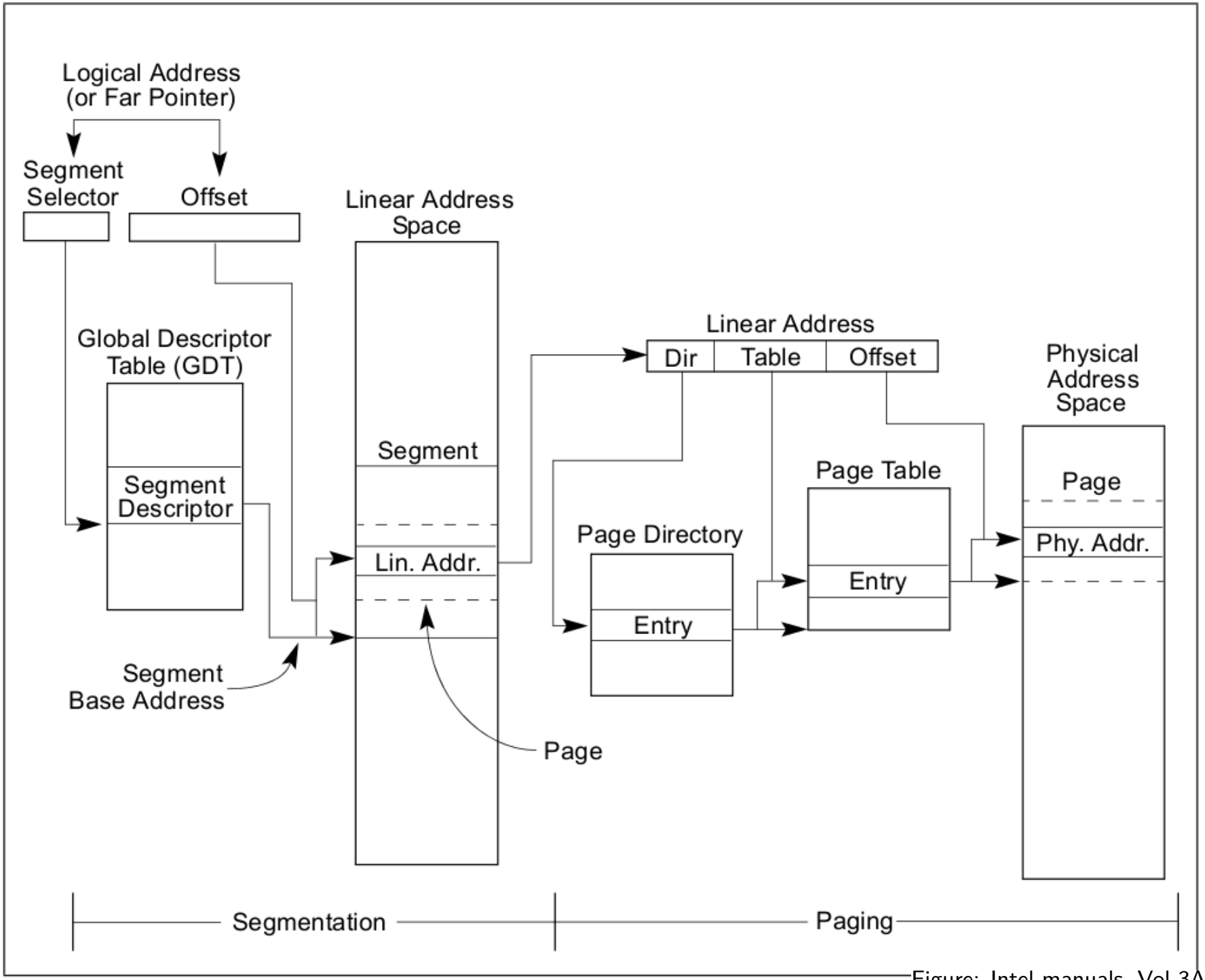


Figure: Intel manuals, Vol 3A

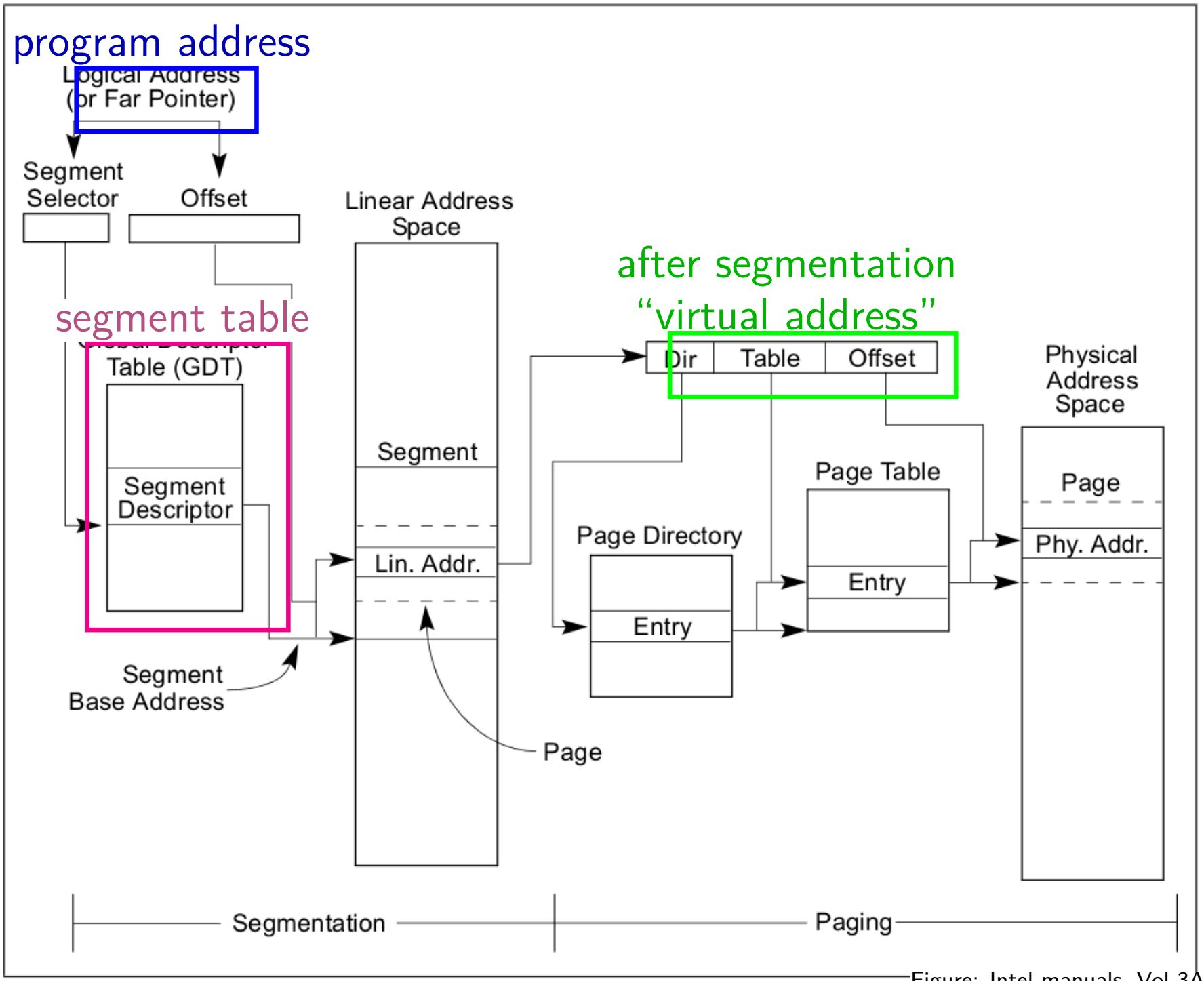


Figure: Intel manuals, Vol 3A

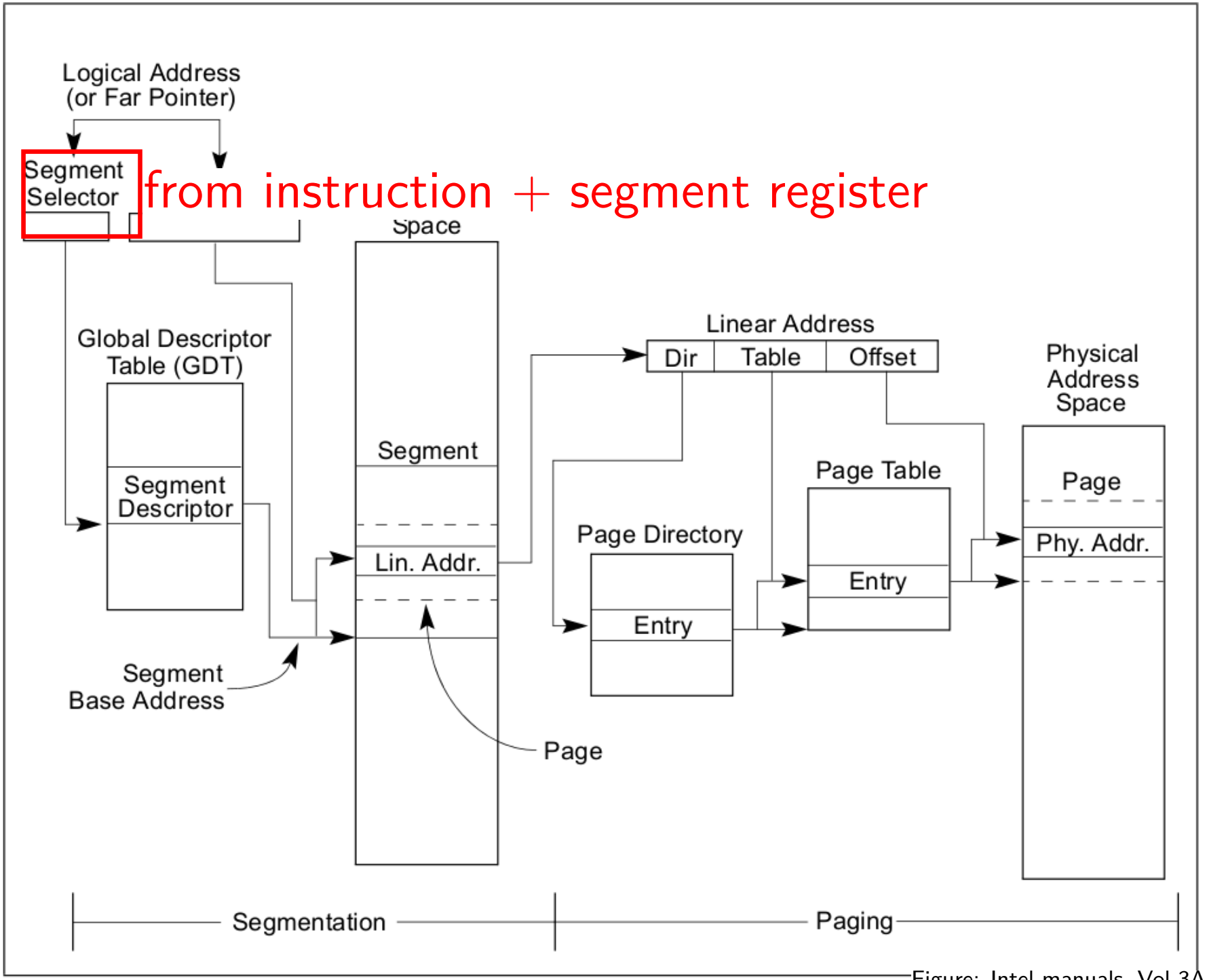


Figure: Intel manuals, Vol 3A

segments and privilege levels

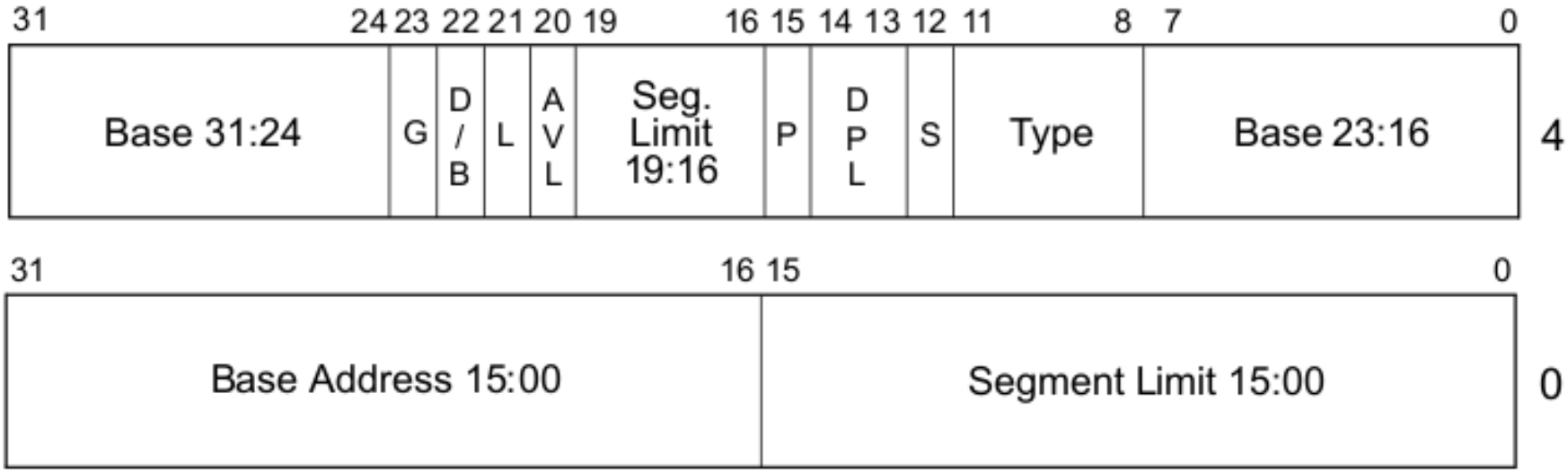
user mode/kernel mode

in x86 — controlled by segment table entry!

64 versus 32-bit mode

in x86 — controlled by segment table entry!

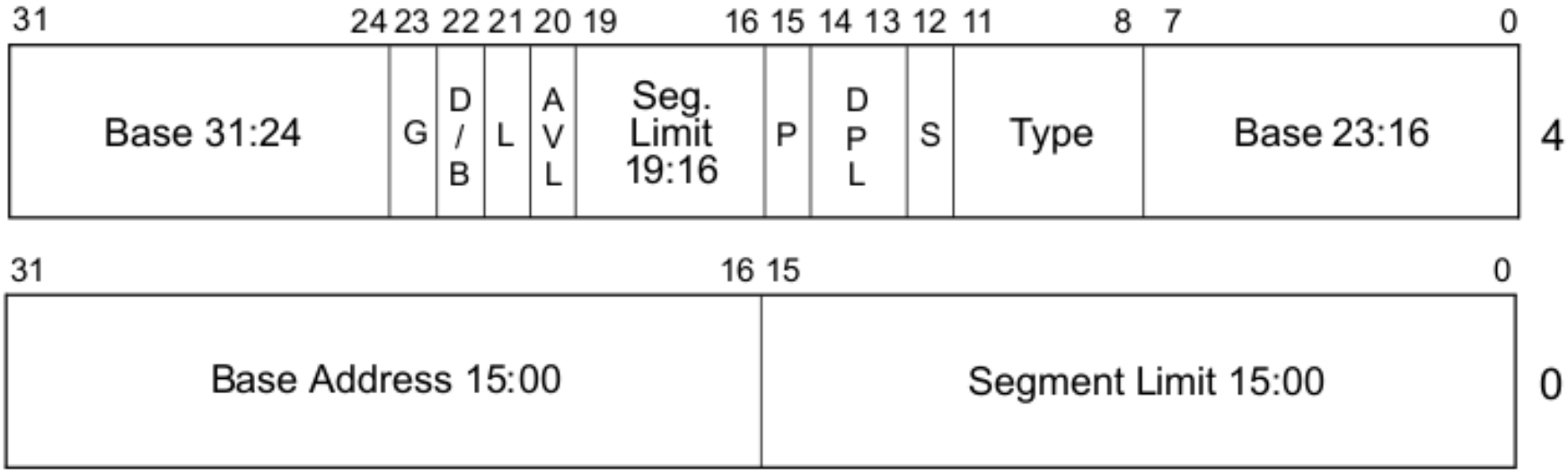
x86 segment descriptor



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure: Intel manuals, Volume 3A

x86 segment descriptor

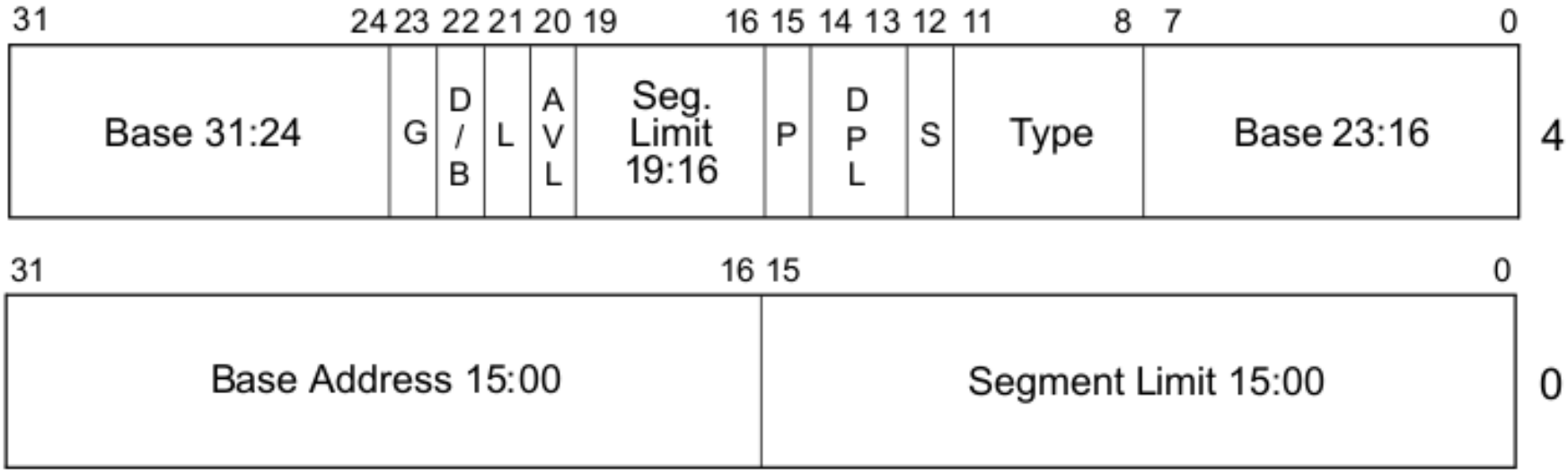


L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

user or kernel mode? (if code)

Figure: Intel manuals, Volume 3A

x86 segment descriptor



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level

64-bit or 32-bit or 16-bit mode? (if code)

LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure: Intel manuals, Volume 3A

64-bit segmentation

in 64-bit mode:

limits are ignored

base addresses are ignored

...except for `%fs`, `%gs`

effectively: extra pointer register

segments on x86

mostly unused even in 32-bit mode

exception: thread-local storage — FS or GS acts as pointer!

reading assembly guides

the calling convention is your friend

- identify functions, arguments, purpose, etc.

don't be scared of the manuals

- or alternate resources
- if you don't know what an instruction is...

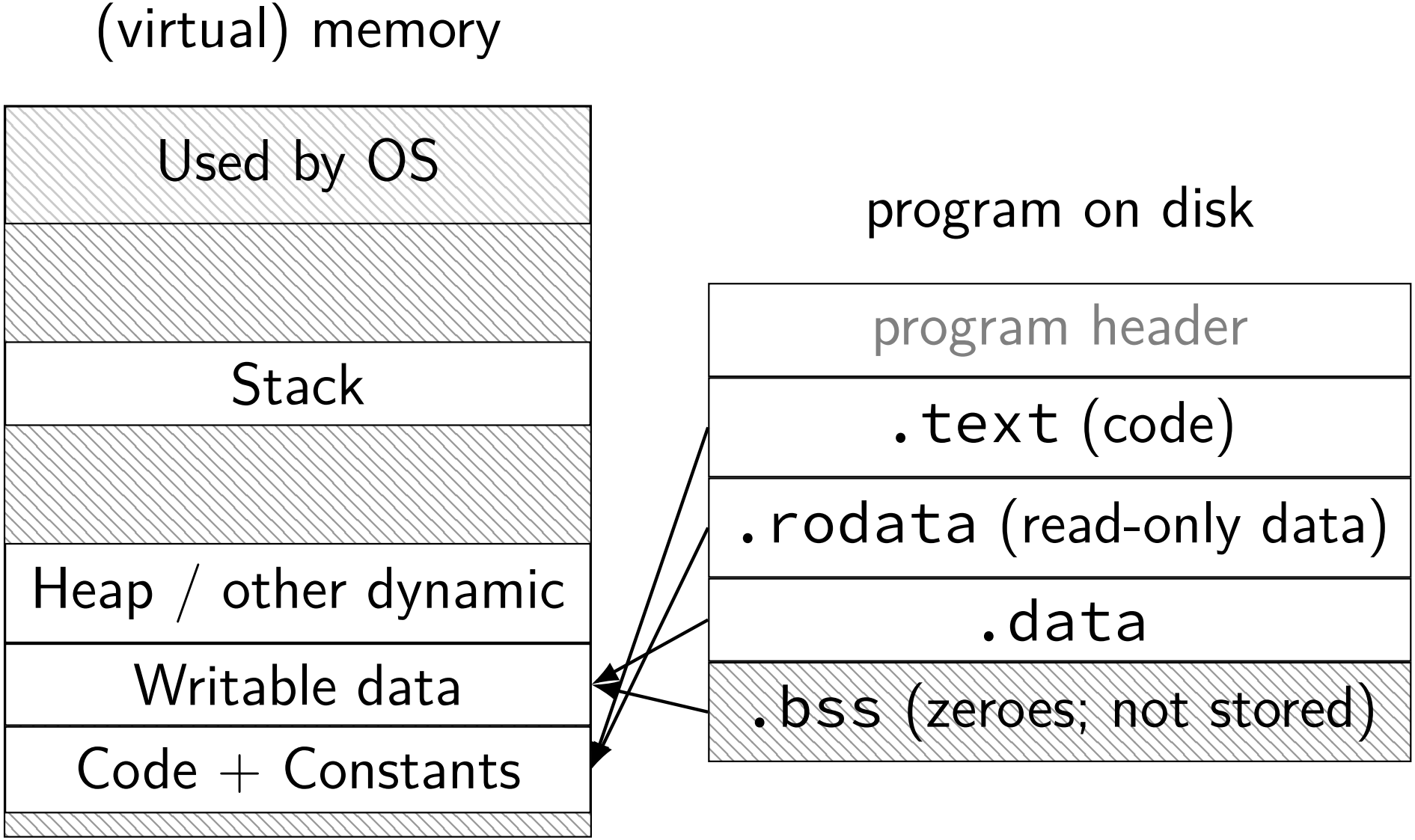
next: machine code

what's actually in binaries

machine code

data about machine code

memory v. disk



ELF (executable and linking format)

Linux (and some others) executable/object file format

header: machine type, file type, etc.
program header: "segments" to load (also, some other information)
segment 1 data
segment 2 data

section header: list of "sections"(mostly for linker)

segments versus sections?

note: ELF terminology; may not be true elsewhere!

sections — exist in object files (and usually executables), used by linker

- have information on intended purpose
- linkers combine these to create executables
- linkers might omit unneeded sections

segments — exist in executables, used to load program

- program loader is dumb — doesn't know what segments are for

ELF example

`objdump -x /bin/busybox` (on my laptop)

`-x`: output all headers

```
/bin/busybox:     file format elf64-x86-64
/bin/busybox
architecture: i386:x86-64, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x0000000000401750

Program Header:
[...]

Sections:
[...]
```

ELF example

`objdump -x /bin/busybox` (on my laptop)

`-x`: output all headers

```
/bin/busybox:     file format elf64-x86-64
/bin/busybox
architecture: i386:x86-64, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x0000000000401750

Program Header:
[...]

Sections:
[...]
```

ELF example

`objdump -x /bin/busybox` (on my laptop)

`-x`: output all headers

```
/bin/busybox:     file format elf64-x86-64
/bin/busybox
architecture: i386:x86-64, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x0000000000401750

Program Header:
[...]

Sections:
[...]
```

a program header (1)

```
Program Header:
LOAD off    0x0000000 vaddr 0x0400000 paddr 0x0400000 align 2**21
     filesz 0x01db697 memsz 0x01db697 flags r-x
LOAD off    0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**21
     filesz 0x00021ee memsz 0x0007d18 flags rw-
[...]
```

load `0x1db697` bytes:

- from `0x0` bytes into the file
- to memory at `0x40000`
- readable and executable

load `0x21ee` bytes:

- from `0x1dbea8`
- to memory at `0x7dbea8`
- with `0x7d18`–`0x21ee` bytes of zeroes
- readable and writable

a program header (1)

```
Program Header:
LOAD off    0x0000000 vaddr 0x0400000 paddr 0x0400000 align 2**21
     filesz 0x01db697 memsz 0x01db697 flags r-x
LOAD off    0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**21
     filesz 0x00021ee memsz 0x0007d18 flags rw-
[...]
```

load `0x1db697` bytes:

- from `0x0` bytes into the file
- to memory at `0x40000`
- readable and executable

load `0x21ee` bytes:

- from `0x1dbea8`
- to memory at `0x7dbea8`
- with `0x7d18`–`0x21ee` bytes of zeroes
- readable and writable

a program header (1)

```
Program Header:
LOAD off    0x0000000 vaddr 0x0400000 paddr 0x0400000 align 2**21
     filesz 0x01db697 memsz 0x01db697 flags r-x
LOAD off    0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**21
     filesz 0x00021ee memsz 0x0007d18 flags rw-
[...]
```

load `0x1db697` bytes:

from `0x0` bytes into the file

to memory at `0x40000`

readable and executable

load `0x21ee` bytes:

from `0x1dbea8`

to memory at `0x7dbea8`

with `0x7d18`–`0x21ee` bytes of zeroes

readable and writable

a program header (1)

```
Program Header:
LOAD off    0x0000000 vaddr 0x0400000 paddr 0x0400000 align 2**21
     filesz 0x01db697 memsz 0x01db697 flags r-x
LOAD off    0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**21
     filesz 0x00021ee memsz 0x0007d18 flags rw-
[...]
```

load `0x1db697` bytes:

- from `0x0` bytes into the file
- to memory at `0x40000`
- readable and executable

load `0x21ee` bytes:

- from `0x1dbea8`
- to memory at `0x7dbea8`
- with `0x7d18`–`0x21ee` bytes of zeroes
- readable and writable

a program header (2)

```
Program Header:
 NOTE off    0x0000190 vaddr 0x0400190 paddr 0x0400190 align 2**2
      filesz 0x0000044 memsz 0x0000044 flags r--
  TLS  off    0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**3
      filesz 0x0000030 memsz 0x000007a flags r--
STACK off    0x0000000 vaddr 0x0000000 paddr 0x0000000 align 2**4
      filesz 0x0000000 memsz 0x0000000 flags rw-
RELRO off    0x01dbea8 vaddr 0x07dbea8 paddr 0x07dbea8 align 2**0
      filesz 0x0000158 memsz 0x0000158 flags r--
```

NOTE — comment

TLS — thread-local storage region (used via `%fs`)

STACK — indicates stack is read/write

RELRO — make this read-only after runtime linking

a section header

```
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .note.ABI-tag 00000020  0000000000400190  0000000000400190  00000190  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.gnu.build-id 00000024  00000000004001b0  00000000004001b0  000001b0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .rela.plt     00000210  00000000004001d8  00000000004001d8  000001d8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .init         0000001a  00000000004003e8  00000000004003e8  000003e8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  4 .plt          00000160  0000000000400410  0000000000400410  00000410  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  5 .text         0017ff1d  0000000000400570  0000000000400570  00000570  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  6 __libc_freeres_fn 00002032  0000000000580490  0000000000580490  00180490  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  7 __libc_thread_freeres_fn 0000021b  00000000005824d0  00000000005824d0  001824d0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  8 .fini         00000009  00000000005826ec  00000000005826ec  001826ec  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  9 .rodata       00044ac8  0000000000582700  0000000000582700  00182700  2**6
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 __libc_subfreeres 000000c0  00000000005c71c8  00000000005c71c8  001c71c8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 11 .stapsdt.base 00000001  00000000005c7288  00000000005c7288  001c7288  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 12 __libc_atexit 00000008  00000000005c7290  00000000005c7290  001c7290  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 13 __libc_thread_subfreeres 00000018  00000000005c7298  00000000005c7298  001c7298  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 14 .eh_frame     000141dc  00000000005c72b0  00000000005c72b0  001c72b0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

sections

tons of “sections”

not actually needed/used to run program

size, file offset, flags (code/data/etc.)

some sections aren’t stored (no “CONTENTS” flag)
— just zeroes

selected sections

`.text`	program code
`.bss`	initially zero data (block started by symbol)
`.data`	other writeable data
`.rodata`	read-only data
`.init`/`.fini`	global constructors/destructors
`.got`/`.plt`	linking related
`.eh_frame`	try/catch related

other executable formats

PE (Portable Executable) — Windows

Mach-O — MacOS X

broadly similar to ELF

differences:

- whether segment/section distinction exists
- how linking/debugging info represented
- how program start info represented

executable startup: naive

copy segments into memory

jump to start address

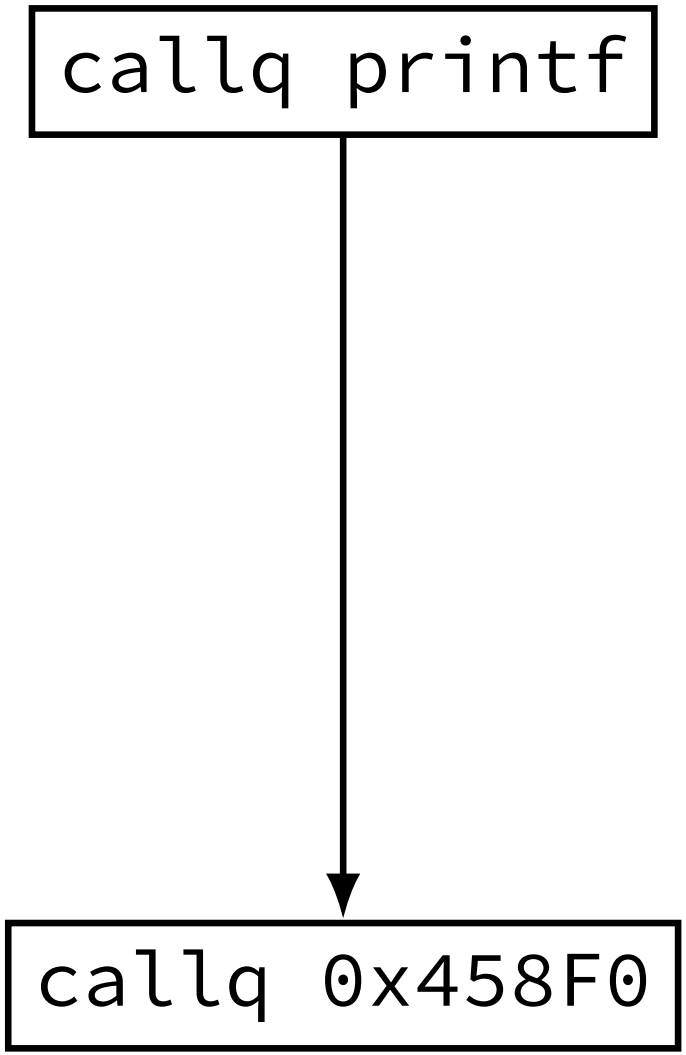
assumes everything is in executable

executable startup: with linking

libraries often seperate from executable

when true: dynamic linking

linking



static v. dynamic linking

static linking — linking done to create executable

dynamic linking — linking done when executable is run

linking data structures

symbol table: `name` $\Rightarrow$ (section, offset)

- example: `main:` in assembly adds symbol table entry for `main`

relocation table: offset $\Rightarrow$ (name, kind)

- example: `call printf` adds relocation for name `printf`

hello.s

```
.data
string: .asciz "Hello, World!"
.text
.globl main
main:
    movq $string, %rdi
    call puts
    ret
```

hello.o

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

hello.o

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

undefined symbol: look for `puts` elsewhere

hello.o

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

insert address of puts, format for `call`

hello.o

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

insert address of string, format for `movq`

hello.o

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

different ways to represent address

32S — signed 32-bit value

PC32 — 32-bit difference from current address

list of kinds in ABI document (with calling conventions)

hello.o

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

g: global — used by other files
l: local

hello.o

```
SYMBOL TABLE:
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l       .data  0000000000000000 string
0000000000000000 g       .text  0000000000000000 main
0000000000000000         *UND*  0000000000000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000003 R_X86_64_32S      .data
0000000000000008 R_X86_64_PC32     puts-0x0000000000000004
```

`.text` segment beginning plus 0 bytes

dynamic versus static linking

dynamic linking could work the same as static linking

but it doesn't

performance issue: avoid changing code

- same code for multiple instances of program
- same memory for filesystem as loaded program

complexity issue: keep OS loader simple

next time: dynamic linking and machine code

RE assignment

assembly reading practice

interlude: strace

`strace` — system call tracer

indicates what system calls (operating system services) used by a program

statically linked hello.exe

`gcc -static -o hello-static.exe hello.s`

`strace ./hello-static.exe`:

```
execve("./hello-static.exe", ["./hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL)                                       = 0x20a5000
brk(0x20a61c0)                                  = 0x20a61c0
arch_prctl(ARCH_SET_FS, 0x20a5880)              = 0
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"..., 4096) =
brk(0x20c71c0)                                  = 0x20c71c0
brk(0x20c8000)                                  = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK)              = -1 ENOENT (No such file or direc
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14)                 = 14
exit_group(14)                                  = ?
+++ exited with 14 +++
```

statically linked hello.exe

```
gcc -static -o hello-static.exe hello.s
```

```
strace ./hello-static.exe:
```

```
execve("./hello-static.exe", ["./hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL)                                       0x20c5000
brk(0x20a61c0)
arch_prctl(ARCH_SET_
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"..., 4096) =
brk(0x20c71c0)                                  = 0x20c71c0
brk(0x20c8000)                                  = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK)              = -1 ENOENT (No such file or direc
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14)                 = 14
exit_group(14)                                  = ?
+++ exited with 14 +++
```

standard library startup

statically linked hello.exe

```
gcc -static -o hello-static.exe hello.s
```

```
strace ./hello-static.exe:
```

```
execve("./hello-static.exe", ["./hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL)                                       0x20a5000
brk(0x20a61c0)                                  0x20a61c0
arch_prctl(ARCH_SET_FS,
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"..., 4096) =
brk(0x20c71c0)                          = 0x20c71c0
brk(0x20c8000)                          = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or direc
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

memory allocation

statically linked hello.exe

`gcc -static -o hello-static.exe hello.s`

`strace ./hello-static.exe`:

```
execve("./hello-static.exe", ["./hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL)
brk(0x20a61c0)
arch_prctl(ARCH_SET_
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"..., 4096) =
brk(0x20c71c0)                          = 0x20c71c0
brk(0x20c8000)                          = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or direc
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

implementation of puts

statically linked hello.exe

```
gcc -static -o hello-static.exe hello.s
```

strace ./hello-static.exe:

```
execve("./hello-static.exe", ["./hello-static.exe"], [/* 46 vars */]) = 0
uname(sysname="Linux", nodename="reiss-lenovo", ...) = 0
brk(NULL)
brk(0x20a61c0)
arch_prctl(ARCH_SET
readlink("/proc/self/exe", "/home/cr4bd/spring2017/cs4630/sl"..., 4096) =
brk(0x20c71c0)                          = 0x20c71c0
brk(0x20c8000)                          = 0x20c8000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or direc
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
write(1, "Hello, World!\n", 14)         = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

standard library shutdown

dynamically linked hello.exe

`gcc -o hello.exe hello.s`

`strace ./hello.exe`:

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or direc
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, st_mode=S_IFREG|0644, st_size=137808, ...) = 0
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7f
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee91
close(3)                                = 0
...
write(1, "Hello, World!\n", 14)          = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

dynamically linked hello.exe

```
gcc -o hello.exe hello.s
```

```
strace ./hello.exe:
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or direc
open("/etc/
fstat(3, st
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7f
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee91
close(3)                                = 0
...
write(1, "Hello, World!\n", 14)          = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

the standard C library (includes puts

dynamically linked hello.exe

```
gcc -o hello.exe hello.s
```

strace ./hello.exe:

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or direc
open("/etc/
fstat(3, st
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7f
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee91
close(3)                                = 0
...
write(1, "Hello, World!\n", 14)          = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

memory allocation (different method)

dynamically linked hello.exe

```
gcc -o hello.exe hello.s
```

strace ./hello.exe:

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or direc
open("/etc/ld.so
fstat(3, st_mode
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7f
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee91
close(3)                                = 0
...
write(1, "Hello, World!\n", 14)          = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

read standard C library header

dynamically linked hello.exe

```
gcc -o hello.exe hello.s
```

strace ./hello.exe:

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or direc
open("/et
fstat(3,
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7f
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee91
close(3)                                = 0
...
write(1, "Hello, World!\n", 14)          = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

load standard C library (3 = opened file)

dynamically linked hello.exe

`gcc -o hello.exe hello.s`

`strace ./hello.exe`:

```
execve("./hello.exe", ["./hello.exe"], [/* 46 vars */]) = 0
...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or direc
open
fsta
...
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"...
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, ..., 3, 0) = 0x7fdfee54d000
mprotect(0x7fdfee70c000, 2097152, PROT_NONE) = 0
mmap(0x7fdfee90c000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7f
mmap(0x7fdfee912000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7fdfee91
close(3)                                = 0
...
write(1, "Hello, World!\n", 14)          = 14
exit_group(14)                          = ?
+++ exited with 14 +++
```

allocate zero-initialized data segment for C library

dynamic linking

load and link (find address of `puts`) runtime

advantages:

- smaller executables
- easier upgrades
- less memory usage (load one copy of library for multiple programs)

disadvantages:

- library upgrades breaking programs
- programs less compatible between OS versions
- possibly slower

where's the linker

Where's the code that calls `open("...libc.so.6")`?

Could check `hello.exe` — it's not there!

where's the linker

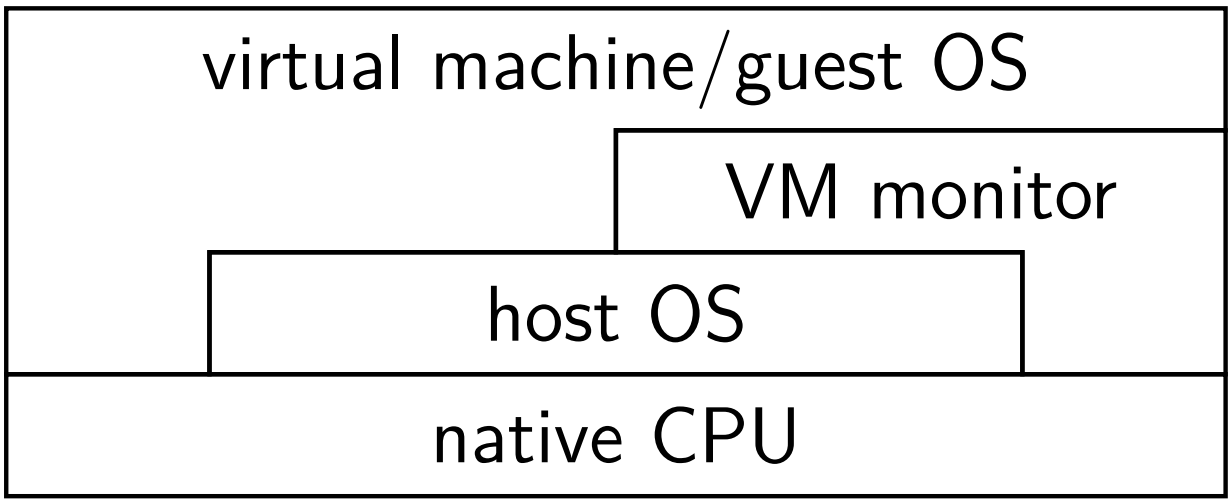
Where's the code that calls `open("...libc.so.6")`?

Could check `hello.exe` — it's not there!

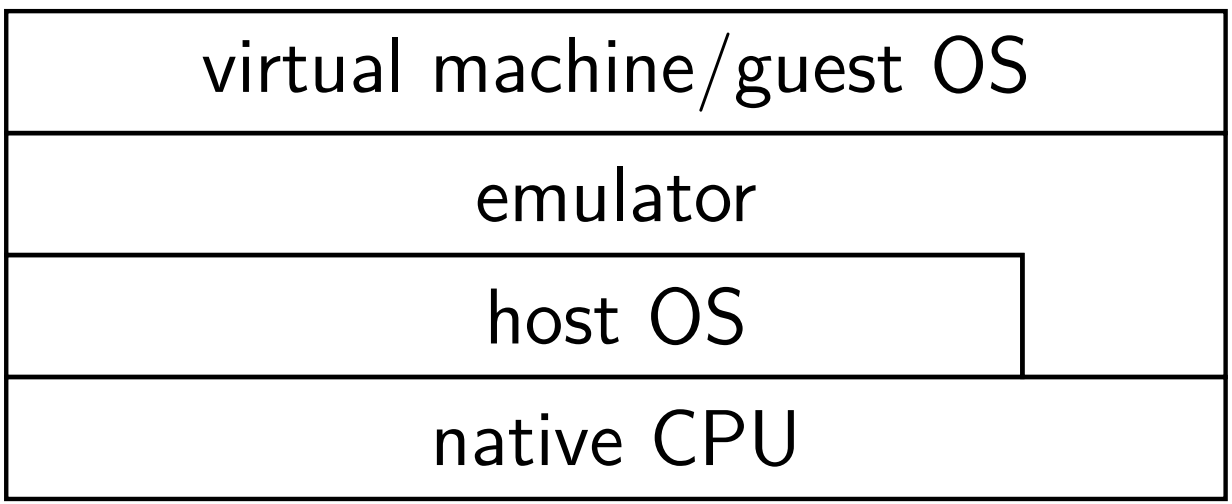
instead: "interpreter" `/lib64/ld-linux-x86-64.so.2`

VM implementation strategies

traditional VM

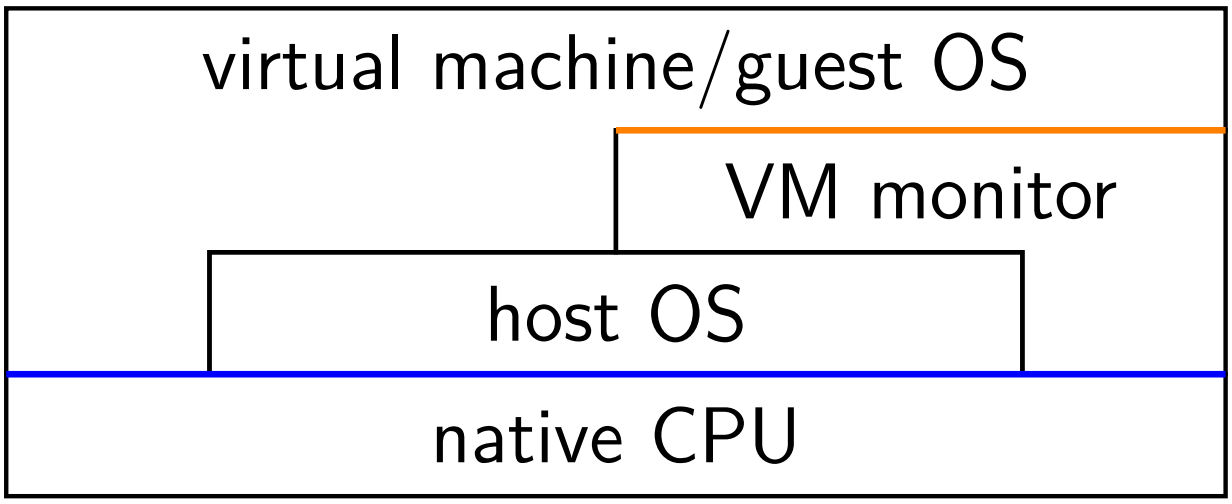


emulator



VM implementation strategies

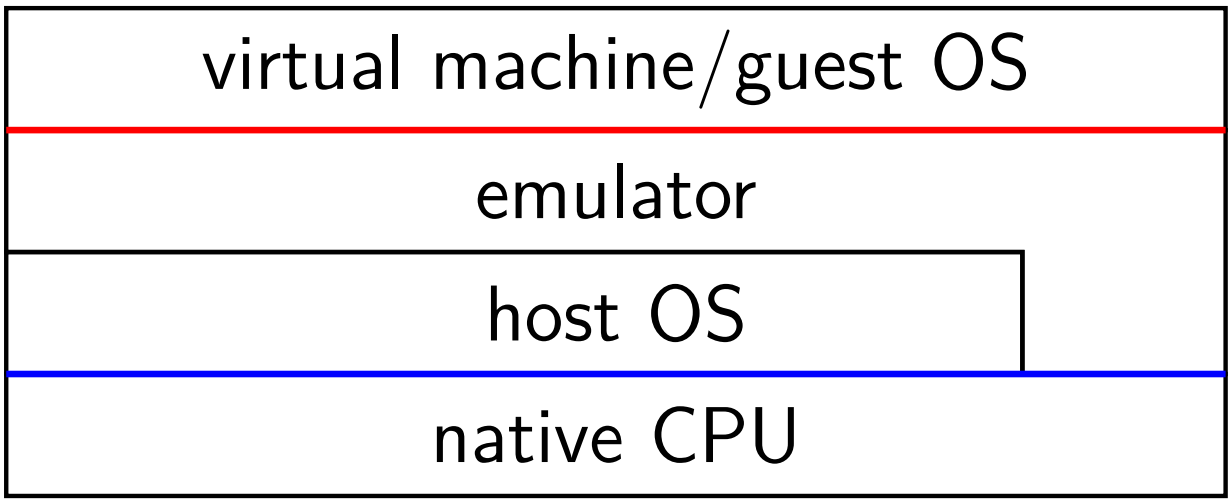
traditional VM



privileged ops become callbacks (help from HW+OS)

native instruction set

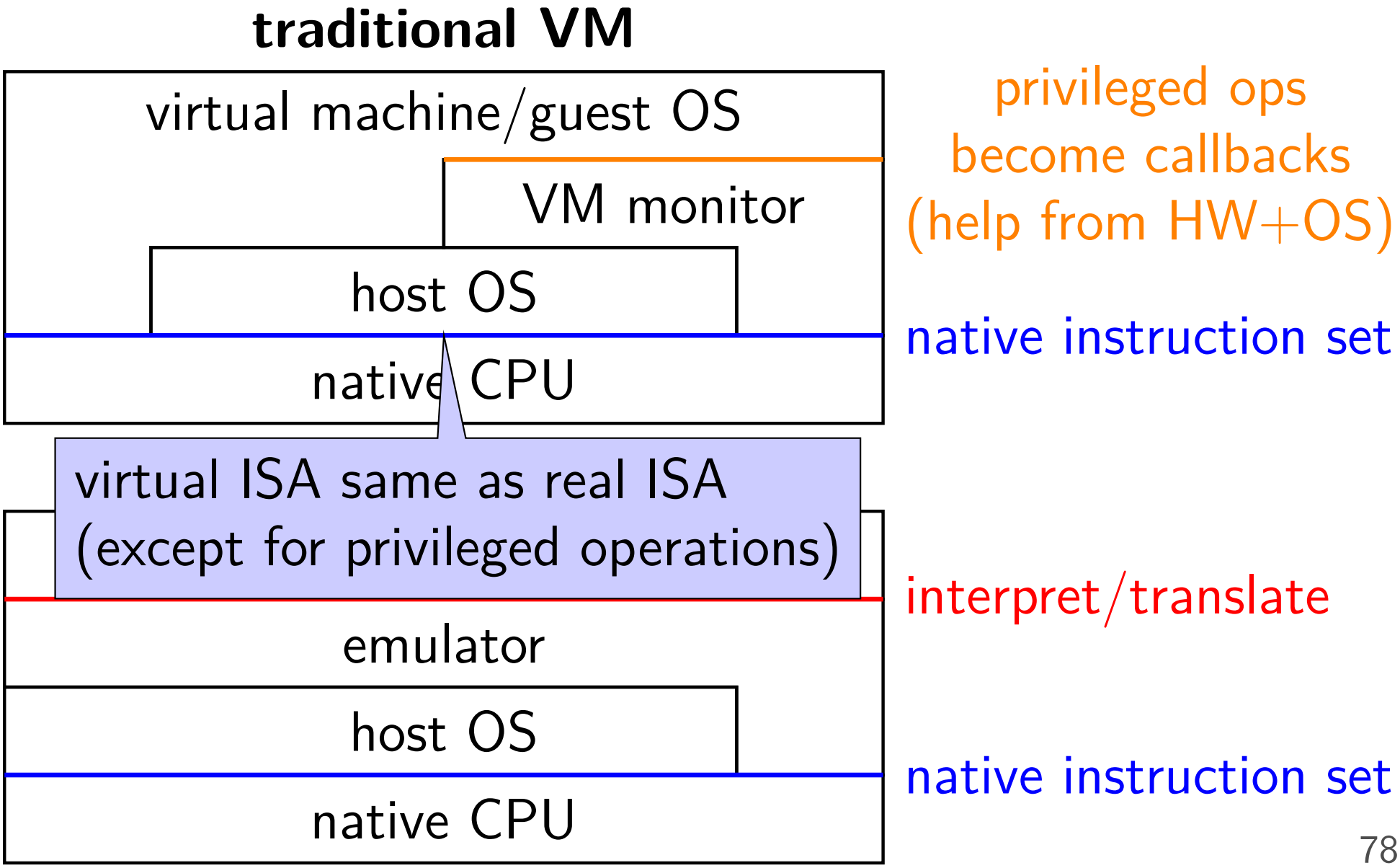
emulator



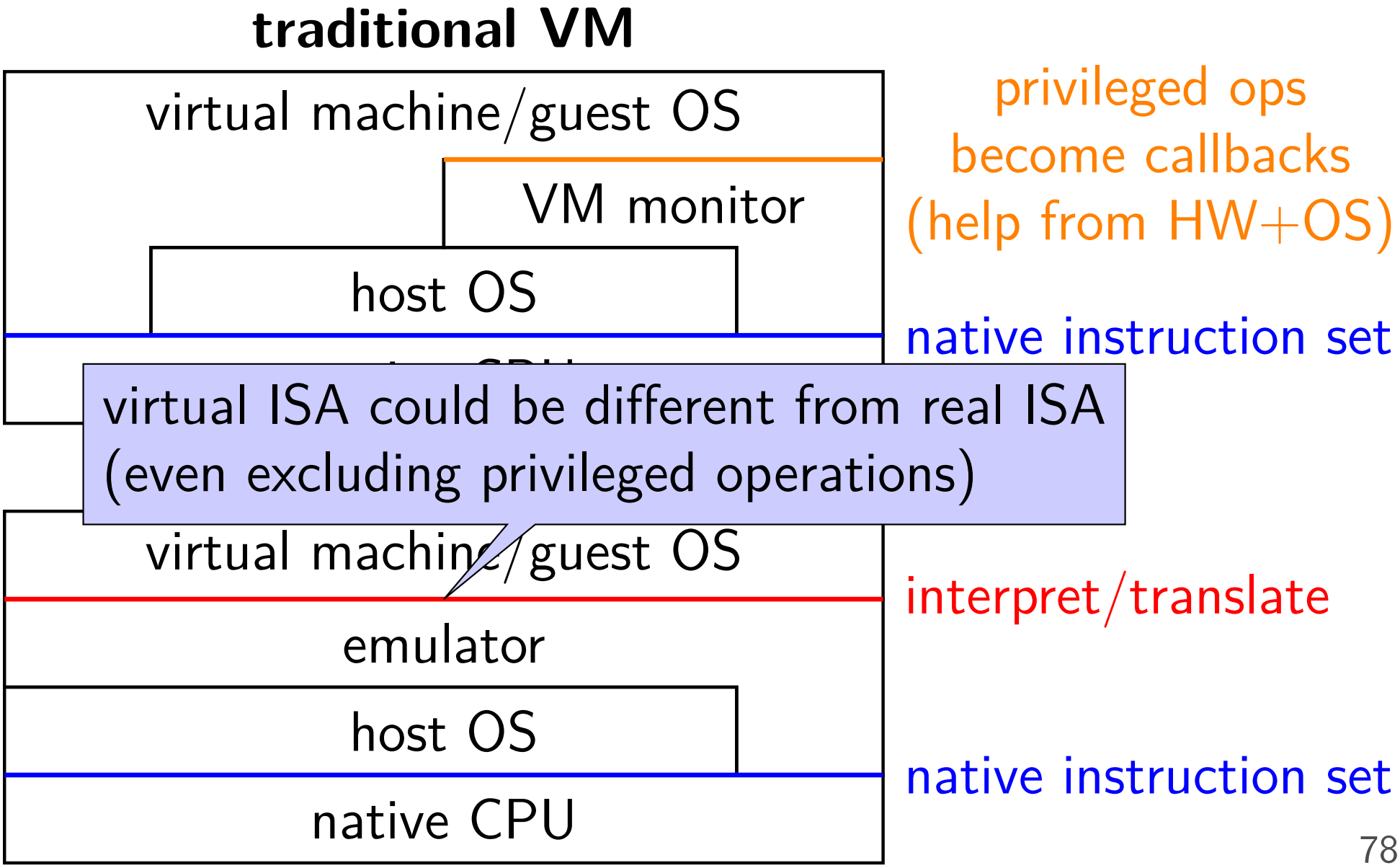
interpret/translate

native instruction set

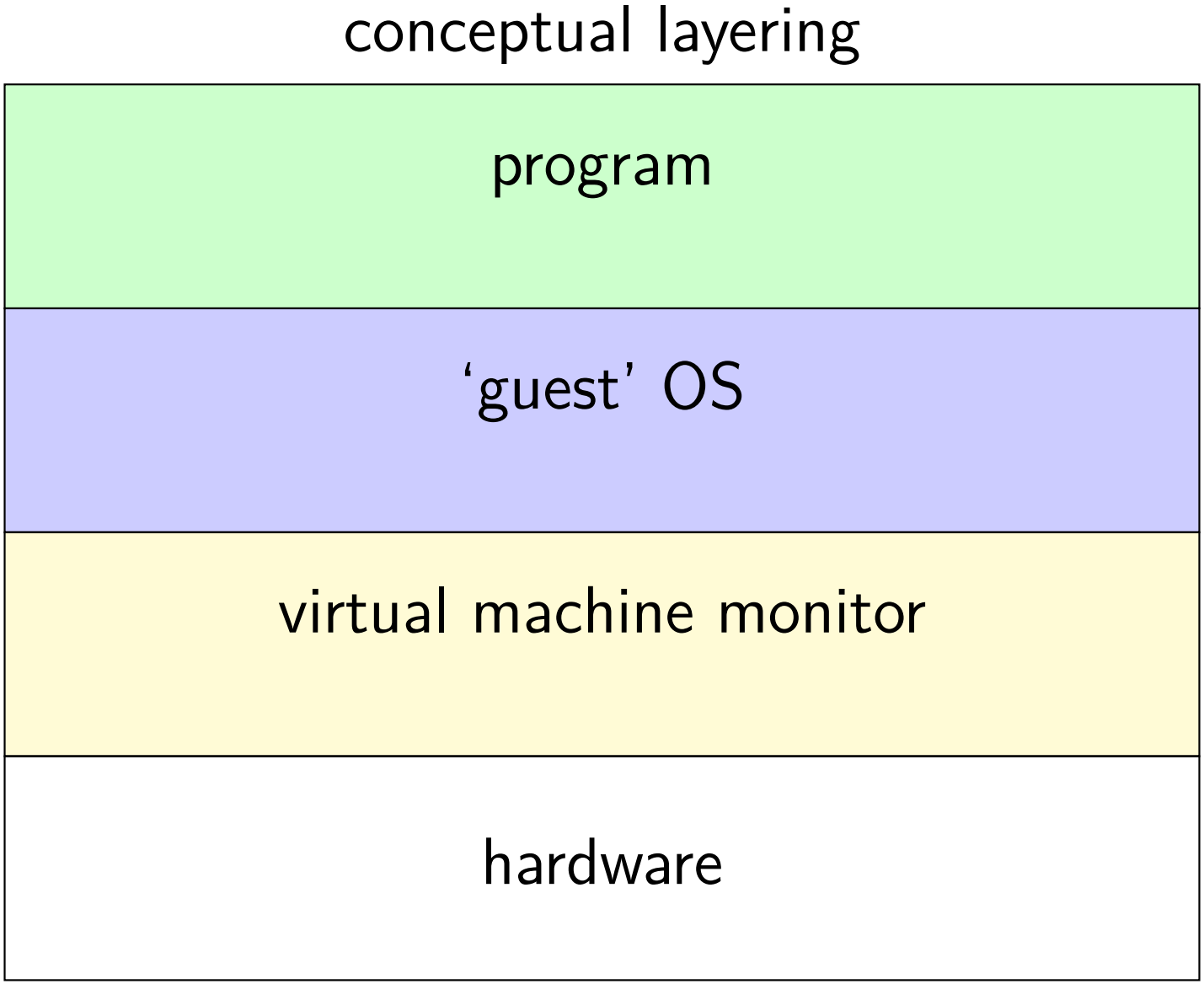
VM implementation strategies



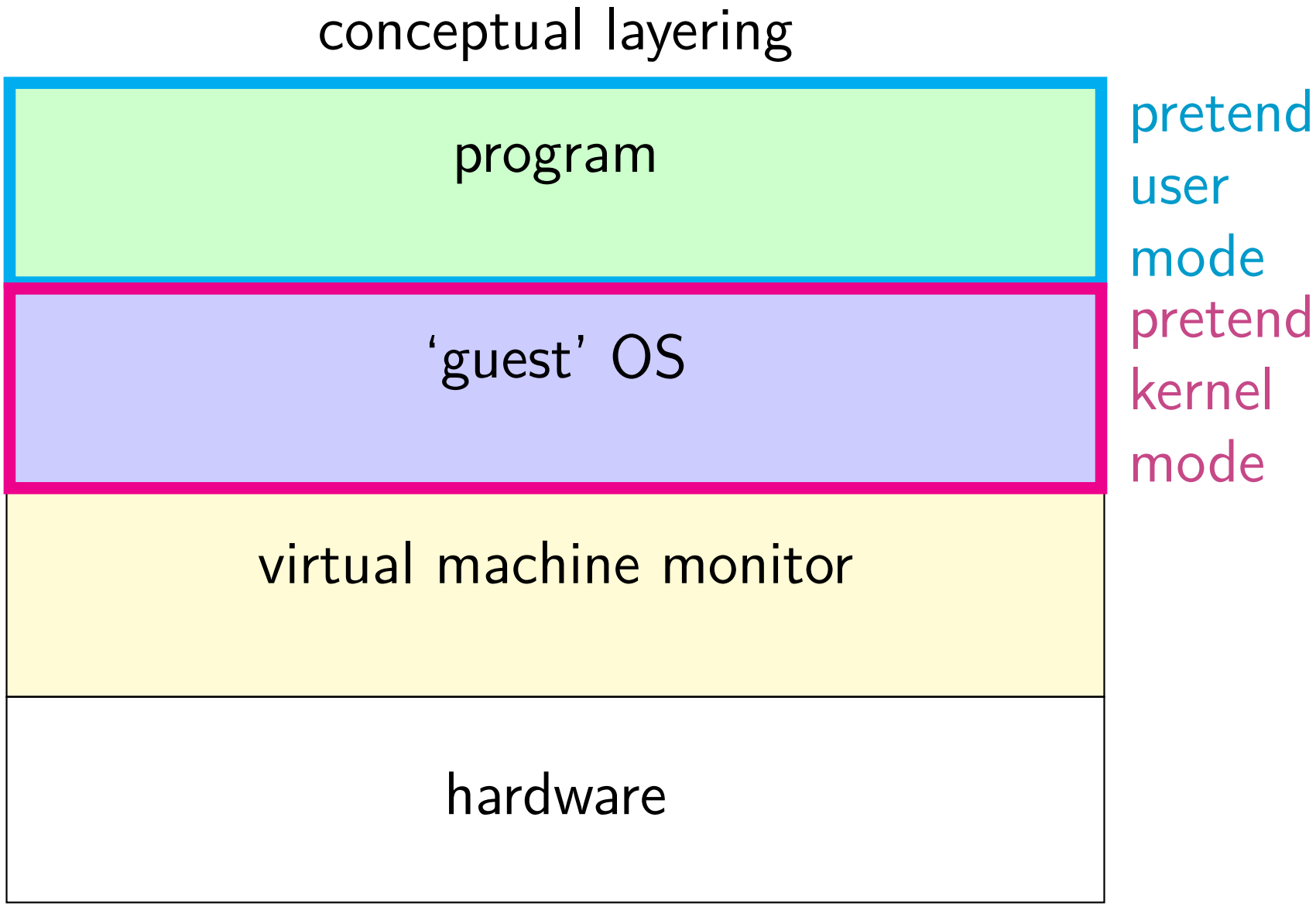
VM implementation strategies



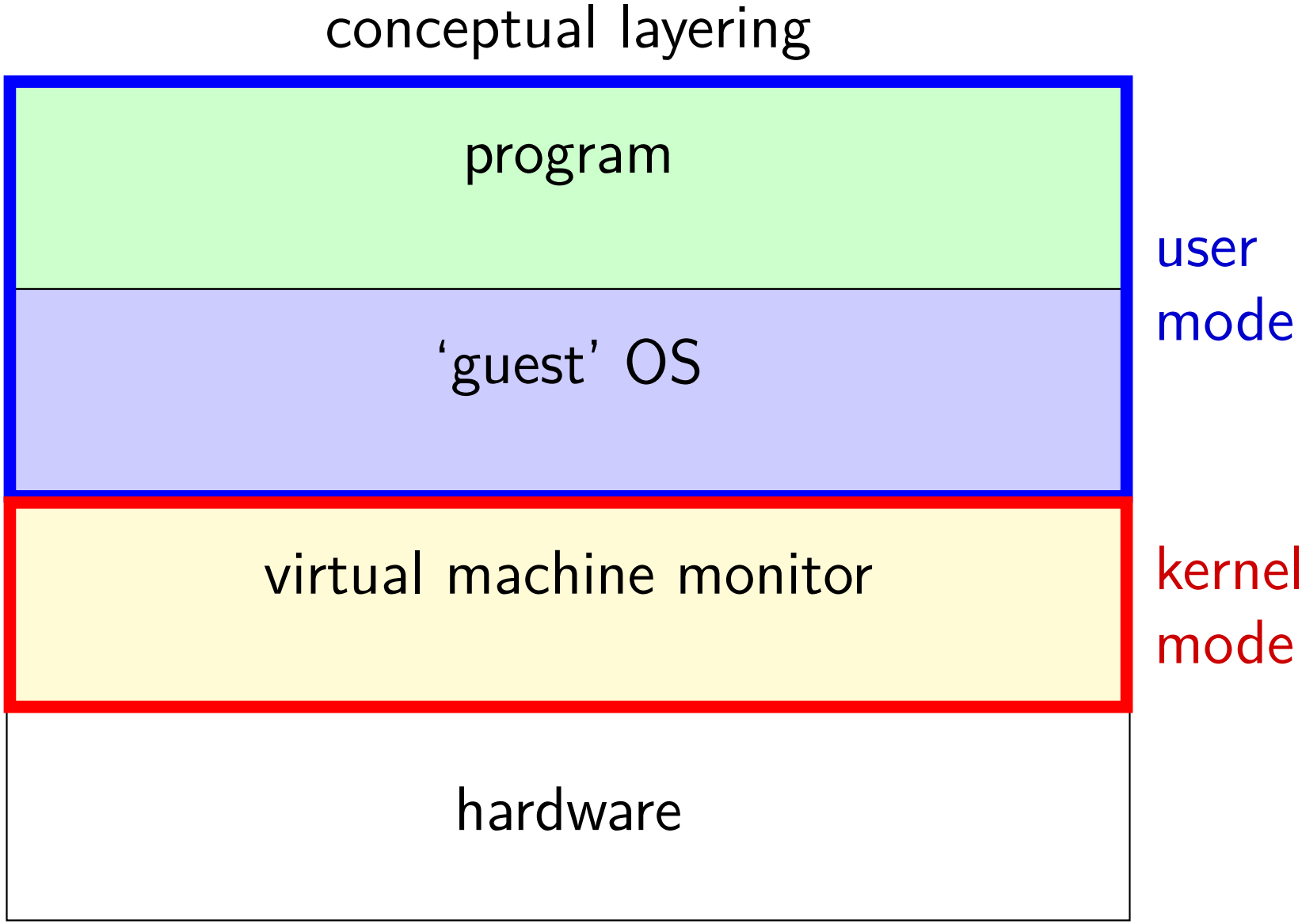
system call flow



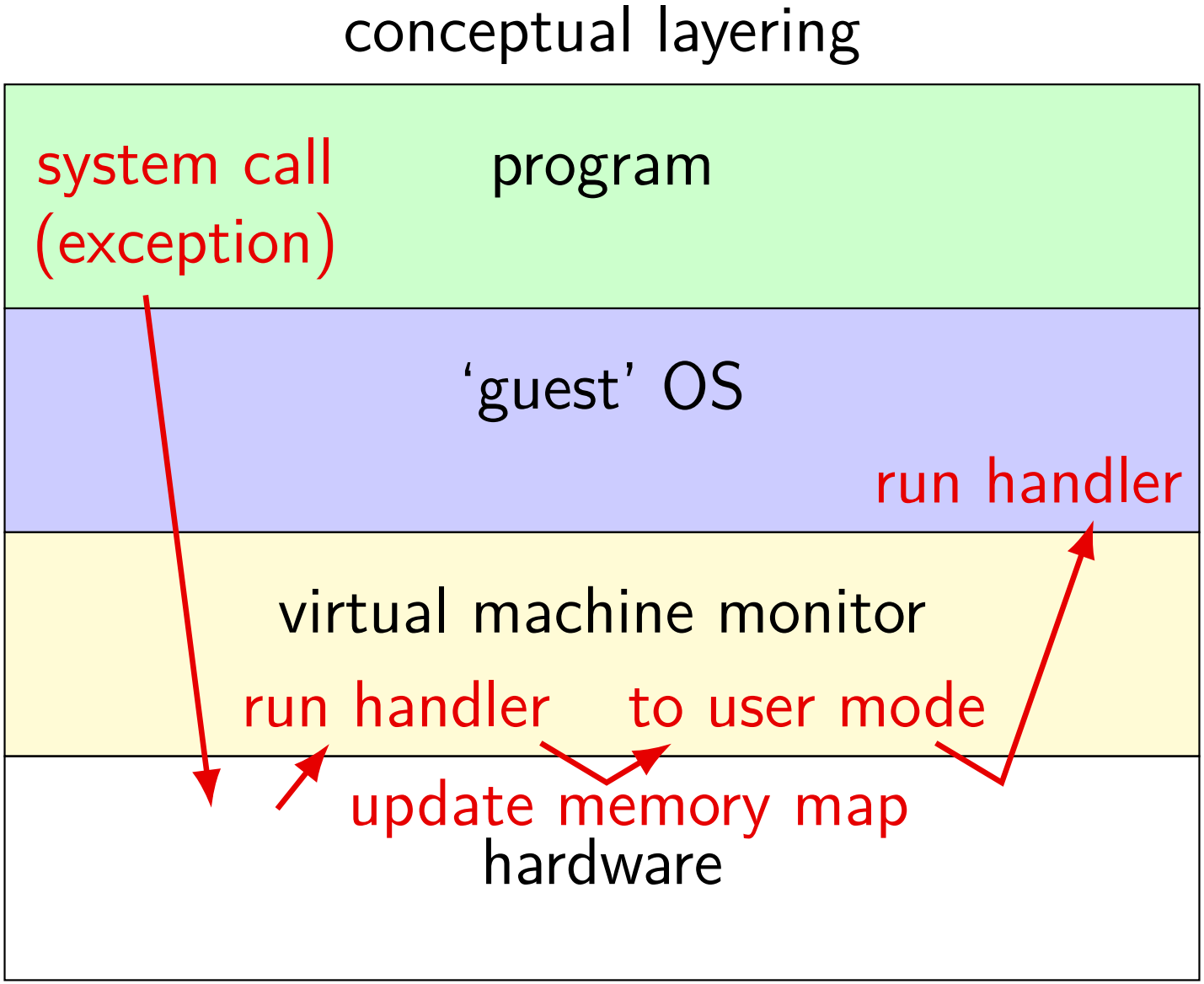
system call flow



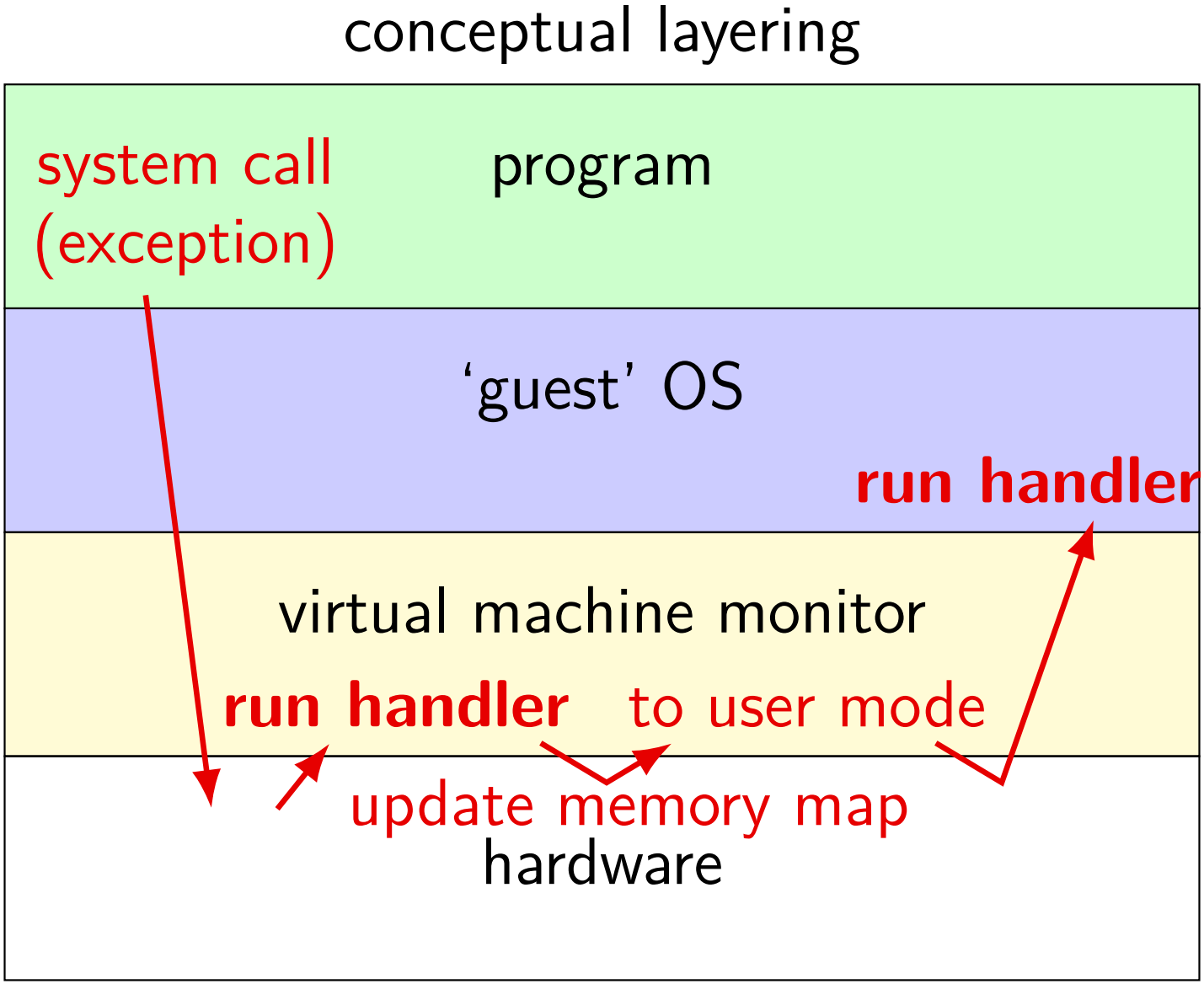
system call flow



system call flow



system call flow



system call flow