virtual machines

last time

access control lists

user and group IDs in processes

set-user-ID programs

briefly: time-of-check-to-time-of-use errors

capabilities: token to address = permission
- token might allow getting other tokens
- can pass between processes
- token specifies type of access (read, write, open files in, kill, …)

minor correction re: POSIX ACLs

implied POSIX ACLs check in order take first/last result

rules are more complicated than that:

- take result for user if any (can prohibit user while allow user's groups)
- take best result for group if any (can prohibit group but allow everyone)
- take default 'other' result otherwise

but designed to allow "do this for group X, with these exceptions"

recall: the virtual machine interface

application

virtual machine interface

operating system

physical machine interface

hardware

system virtual machine
(VirtualBox, VMWare, Hyper-V, ...)

process virtual machine
(typical operating systems)

imitate physical interface
(of some real hardware)

chosen for convenience
(of applications)

recall: the virtual machine interface

application

virtual machine interface

operating system

physical machine interface

hardware

system virtual machine
(VirtualBox, VMWare, Hyper-V, ...)

process virtual machine
(typical operating systems)

imitate physical interface
(of some real hardware)

chosen for convenience
(of applications)

system virtual machine

goal: imitate hardware interface

what hardware?

- usually — whatever's easiest to emulate

system virtual machine terms

hypervisor or *virtual machine monitor*

something that runs system virtual machines

guest OS

operating system that runs as application on hypervisor

host OS

operating system that runs hypervisor

sometimes, hypervisor is the OS (doesn't run normal programs)

imitate: how close?

full virtualization

- guest OS runs unmodified, as if on real hardware

paravirtualization

- *small* modifications to guest OS to support virtual machine
- might change, e.g., how page table entries are set
- why — we'll talk later

fuzzy line — custom device drivers sometimes not called paravirtualization

multiple techniques

today: talk about one way of implementing VMs

there are some variations I won't mention

...or might not have time to mention

one variation: extra HW support for VMs (if time)

one variation: compile guest OS code to new machine code
- not as slow as you'd think, sometimes

terms for this lecture

virtual address — virtual address for guest OS

physical address — physical address for guest OS

machine address — physical address for hypervisor/host OS

process control block for guest OS

guest OS runs like a process, but…

have extra things for hypervisor to track:

if guest OS thinks interrupts are disabled

what guest OS thinks is it's interrupt handler table

what guest OS thinks is it's page table base register

if guest OS thinks it is running in kernel mode

…

hypervisor basic flow

guest OS operations trigger exceptions

- e.g. try to talk to device: page or protection fault
- e.g. try to disable interrupts: protection fault
- e.g. try to make system call: system call exception

hypervisor exception handler tries to do what processor would "normally" do

- talk to device on guest OS's behalf
- change "interrupt disabled" flag for hypervisor to check later
- invoke the guest OS's system call exception handler

virtual machine execution pieces

making IO and kernel-mode-related instructions work

- solution: trap-and-emulate
- force instruction to cause fault
- make fault handler do what instruction would do
- might require reading machine code to emulate instruction

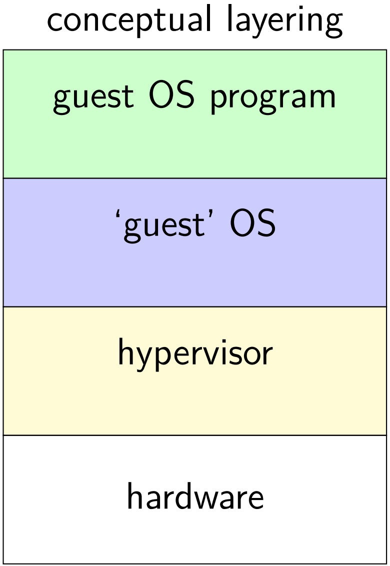
making exceptions/interrupts work

- 'reflect' exceptions/interrupts into guest OS
- same setup processor would do …
- but do setup on guest OS registers + memory

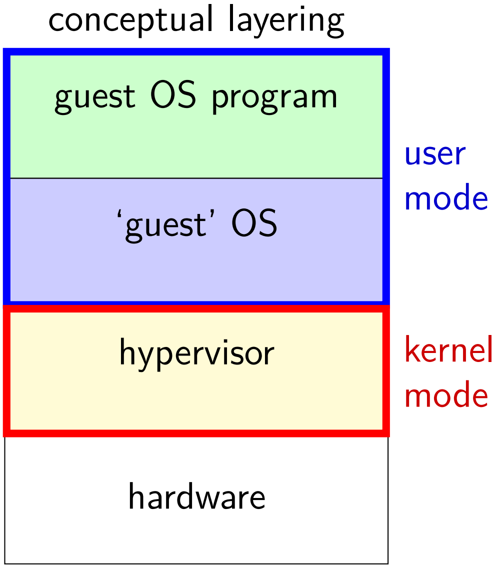
making page tables work

- it's own topic

VM layering (intro)

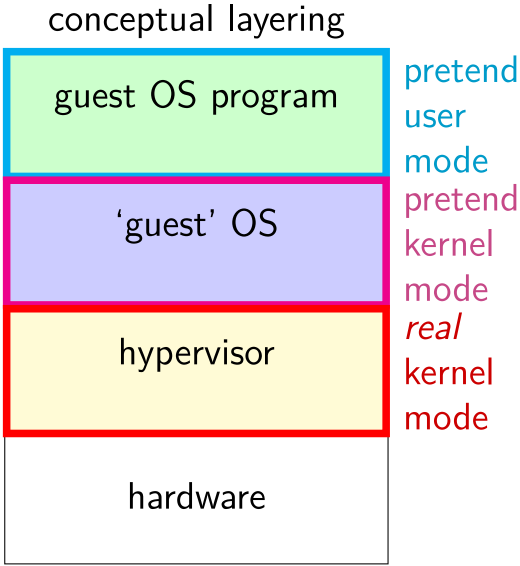


VM layering (intro)

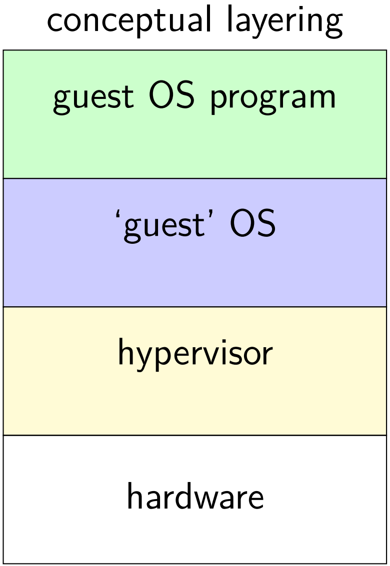


$\approx$ hypervisor's process

VM layering (intro)

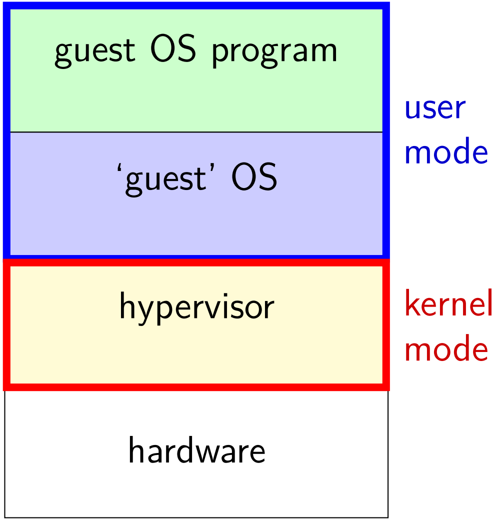


VM layering


conceptual layering
guest OS program
‘guest’ OS
hypervisor
hardware

VM layering

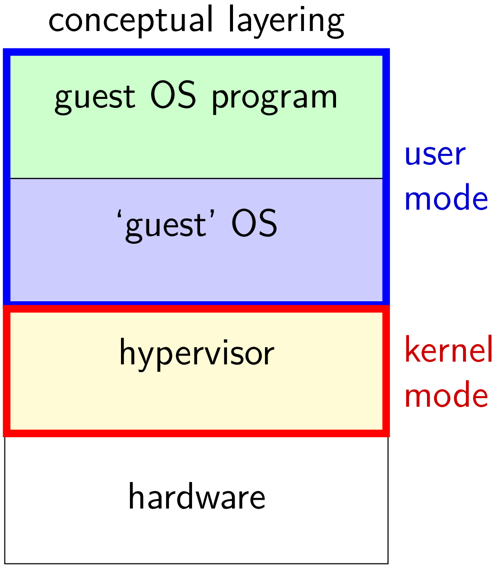
conceptual layering



hypervisor tracks...

guest OS registers
page table: physical to machine addresses
I/O devices guest OS can access
...

VM layering



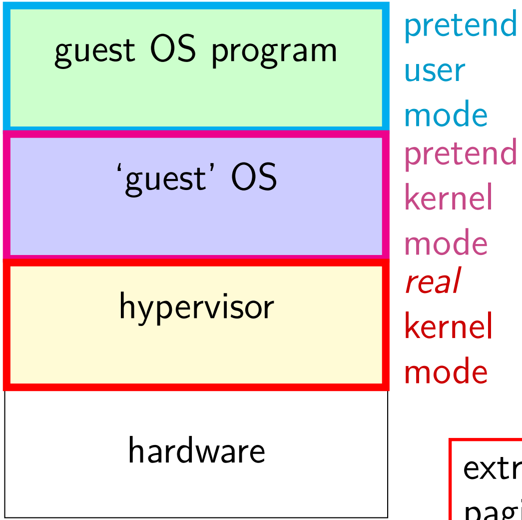
hypervisor tracks...

guest OS registers
page table: physical to machine addresses
I/O devices guest OS can access
...

same as for normal process so far...
(except renamed virtual/physical addrs)

VM layering

conceptual layering



hypervisor tracks...

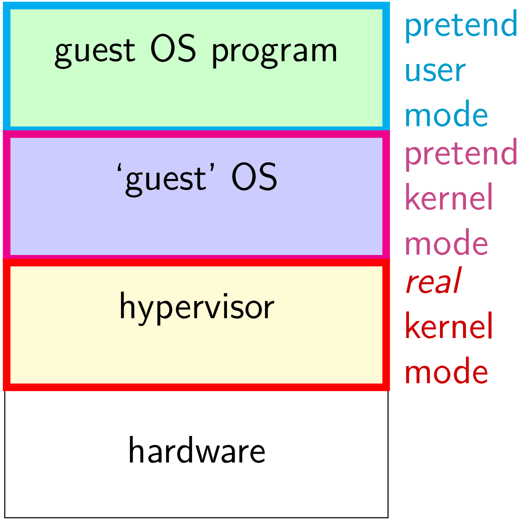
guest OS registers
page table: physical to machine addresses
I/O devices guest OS can access
...

whether in user/kernel mode
guest OS page table ptr (virt to phys)
guest OS exception table ptr
...

extra state to impl. pretend kernel mode
paging, protection, exceptions/interrupts

VM layering

conceptual layering



hypervisor tracks...

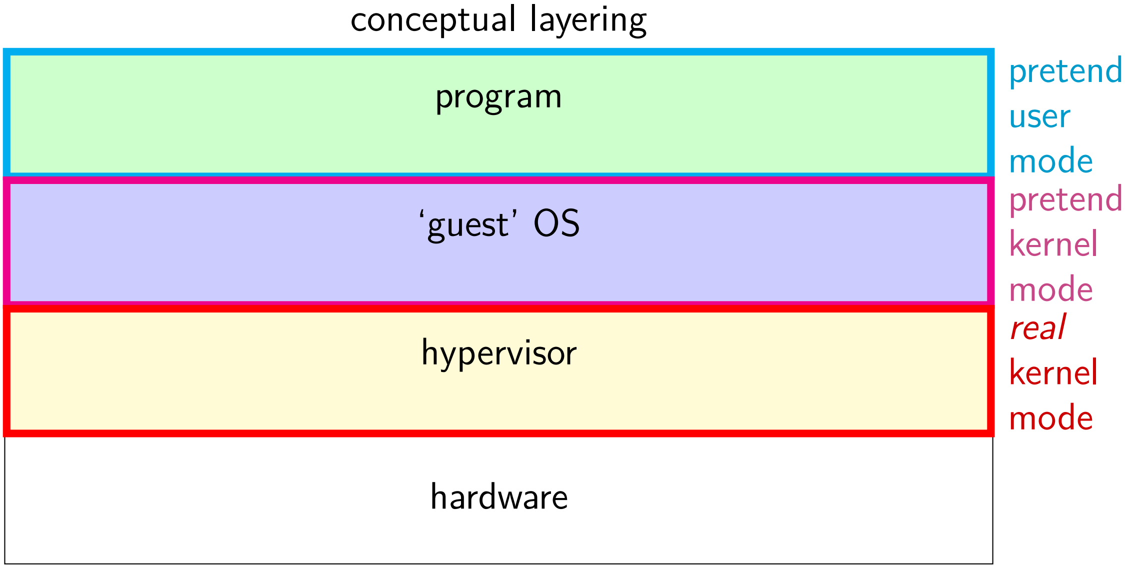
guest OS registers
page table: physical to machine addresses
I/O devices guest OS can access
...

whether in user/kernel mode
guest OS page table ptr (virt to phys)
guest OS exception table ptr
... virtual machine state

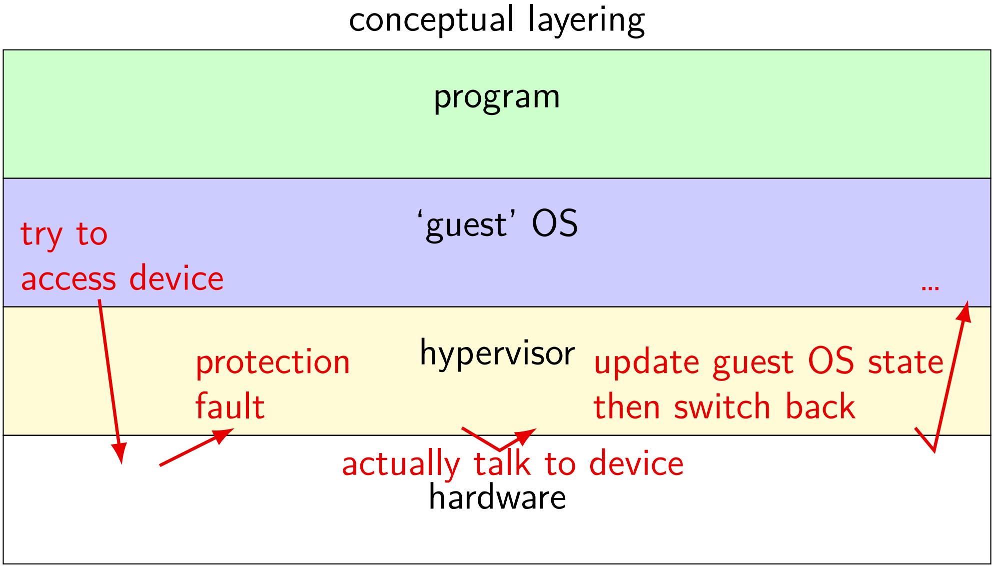
virtual to machine address page table ...

extra data structures to translate pretend kernel mode info to form real CPU understands

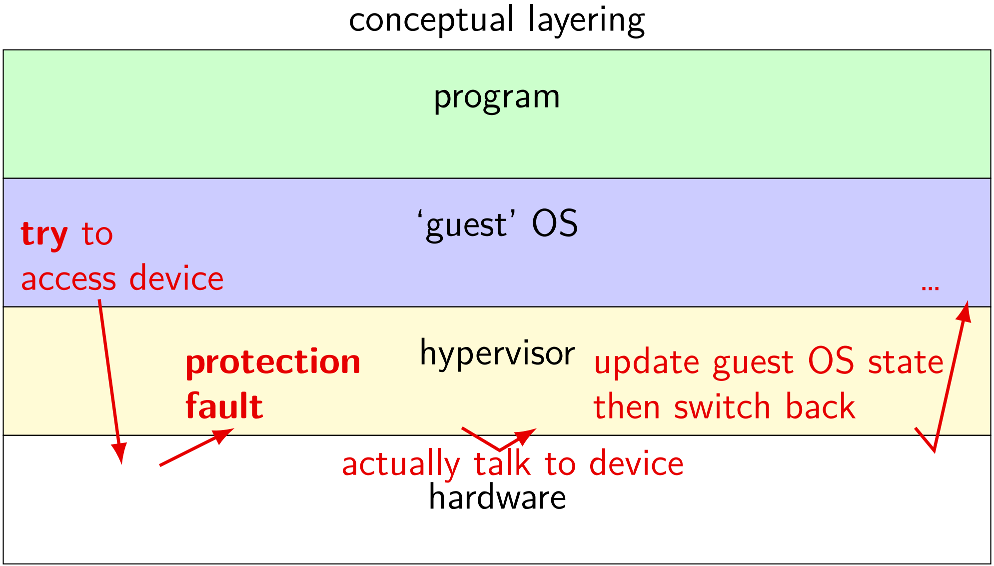
privileged I/O flow



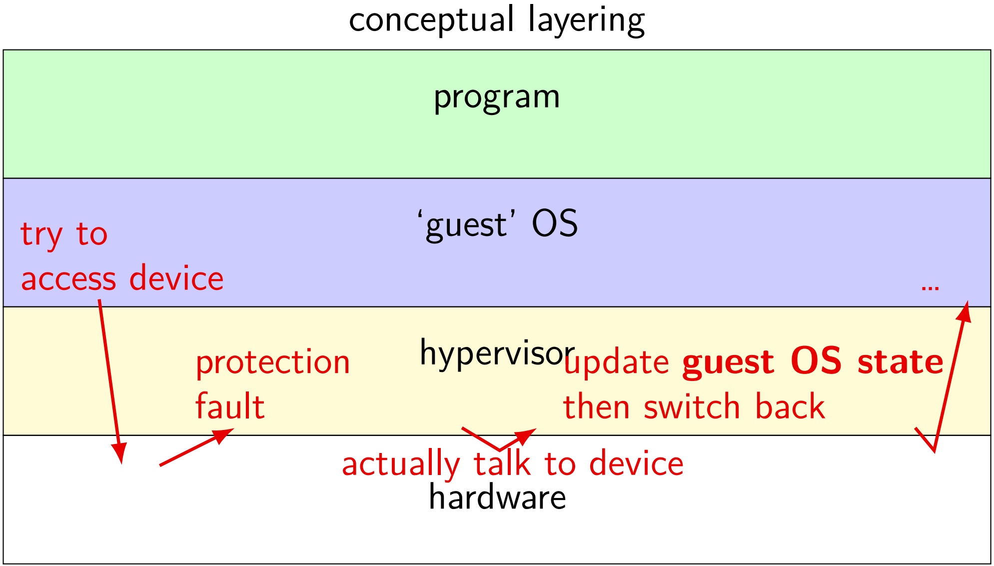
privileged I/O flow



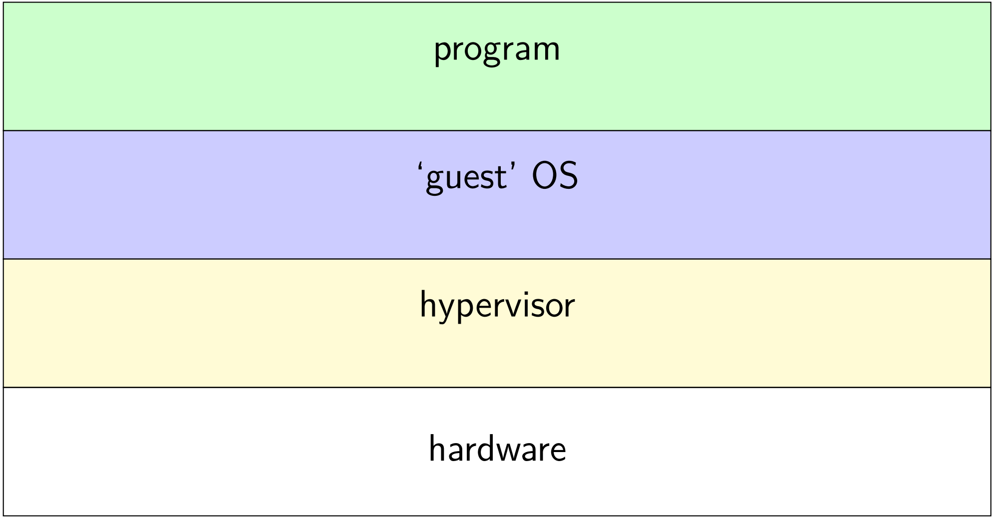
privileged I/O flow



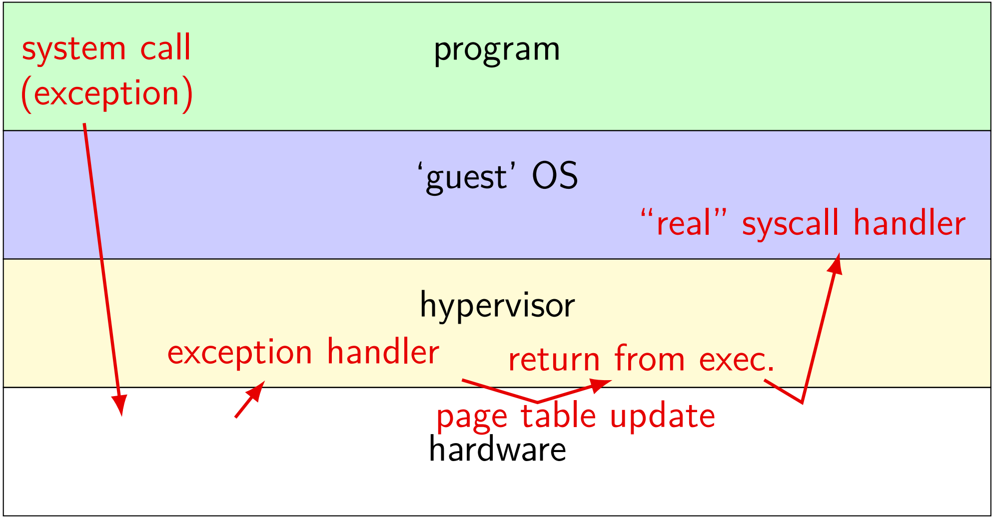
privileged I/O flow



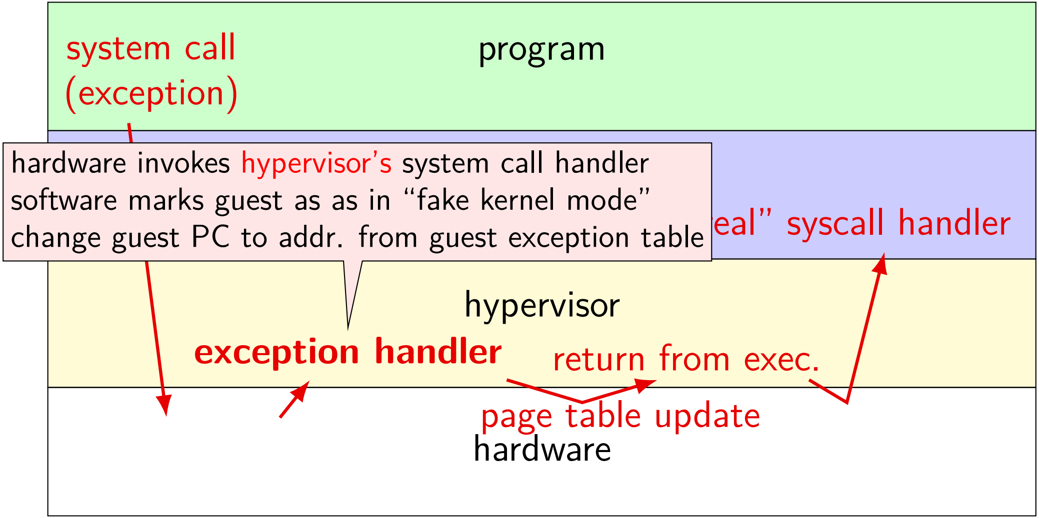
system call/exception flow (part 1)



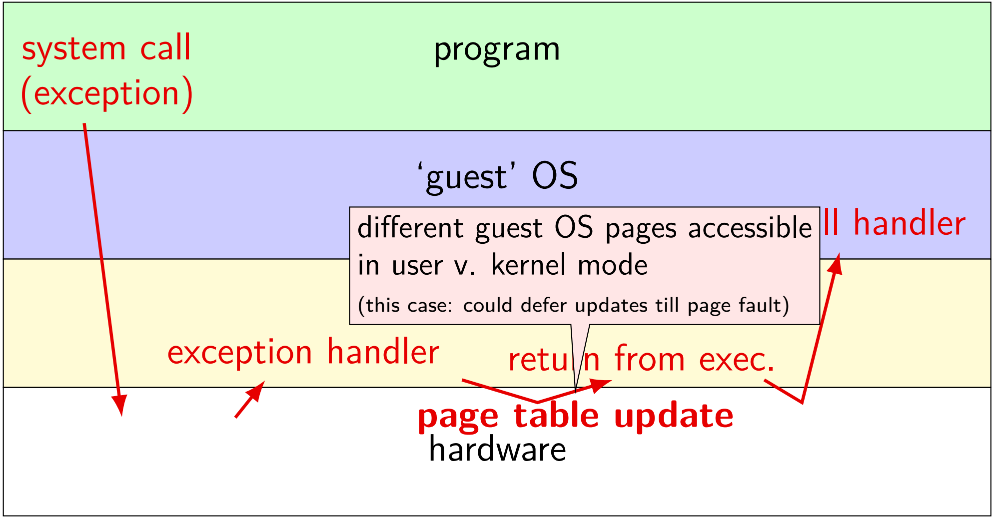
system call/exception flow (part 1)



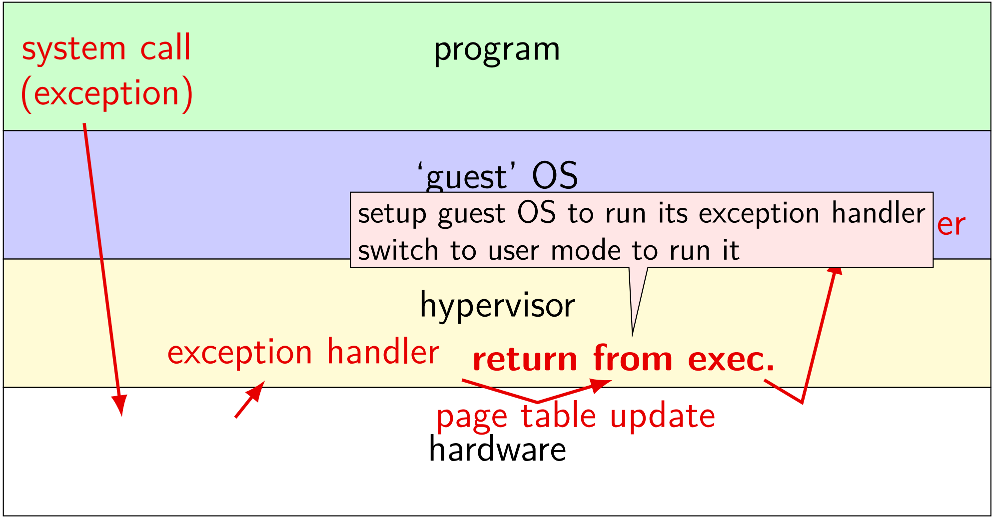
system call/exception flow (part 1)



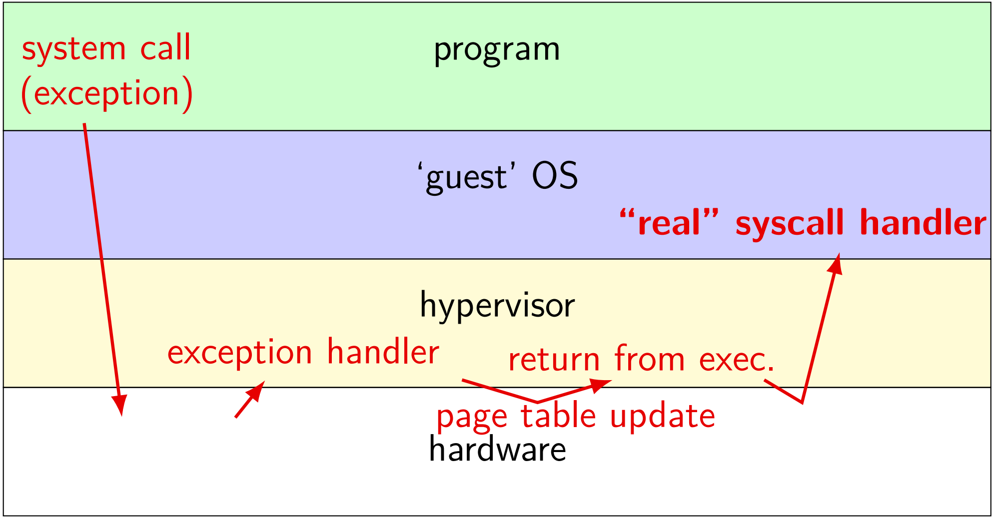
system call/exception flow (part 1)



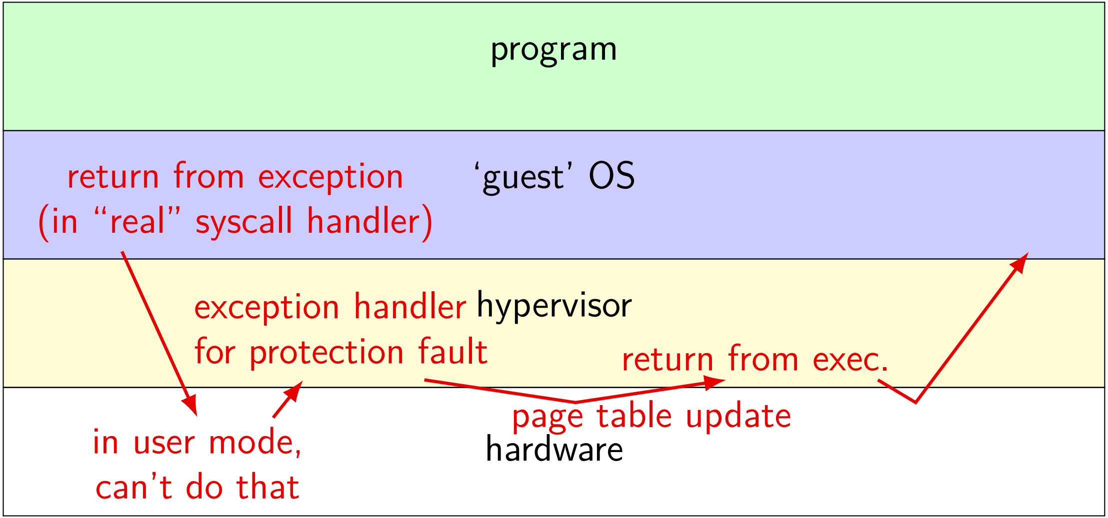
system call/exception flow (part 1)



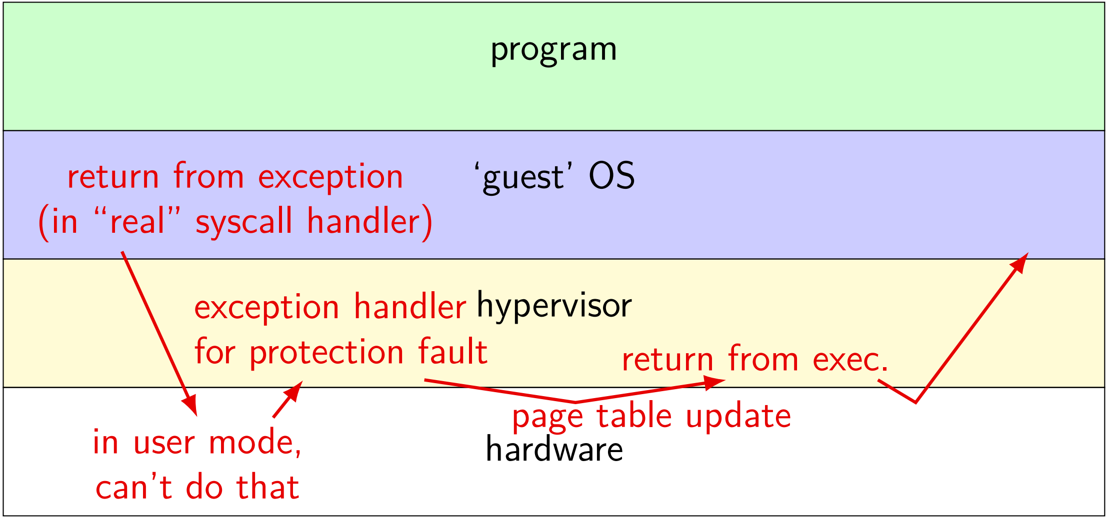
system call/exception flow (part 1)



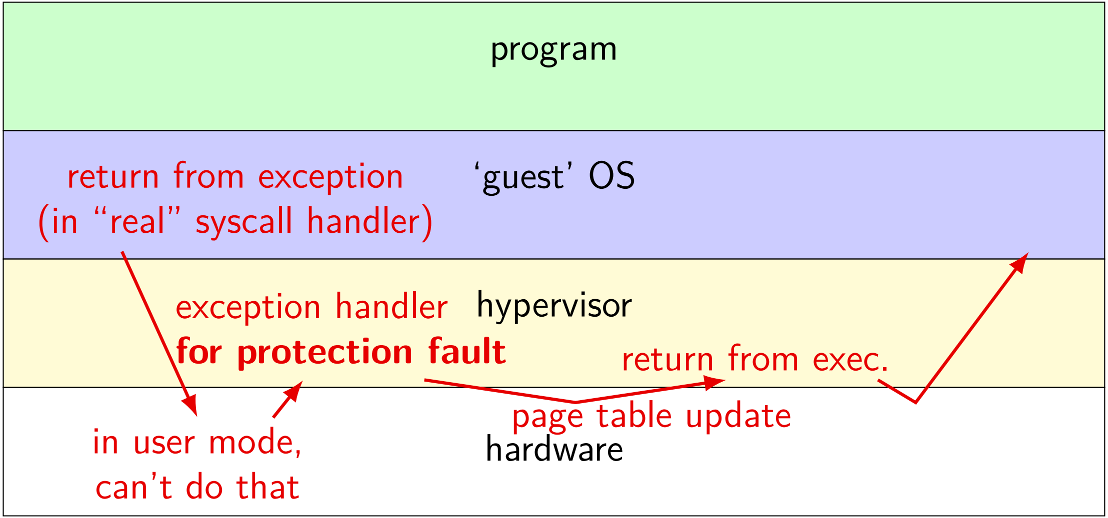
system call/exception flow (part 2)



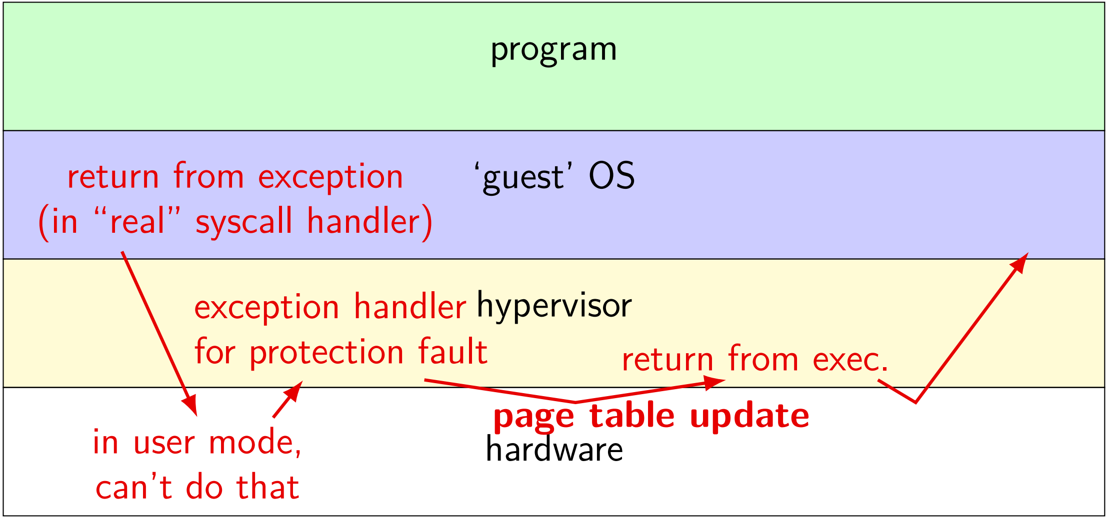
system call/exception flow (part 2)



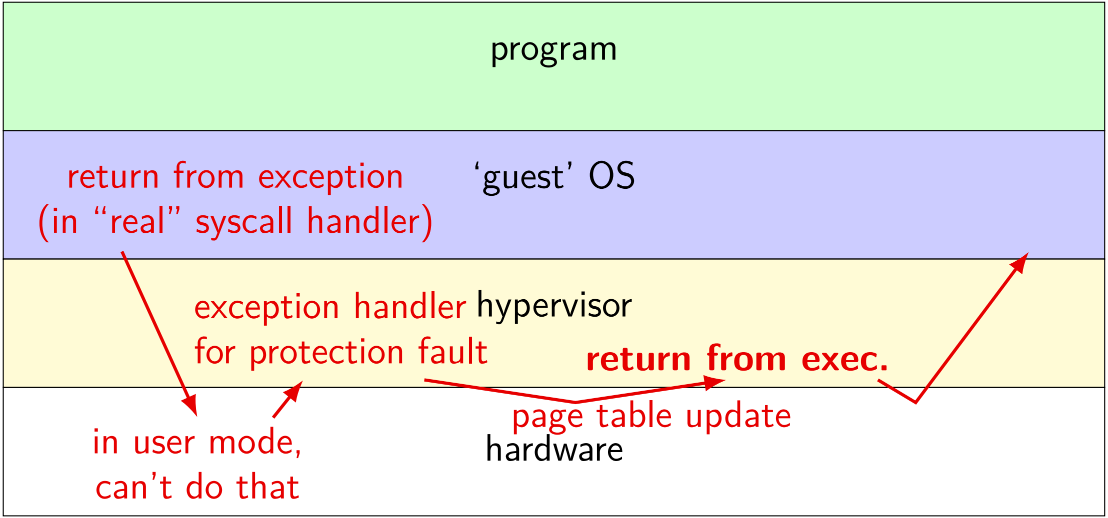
system call/exception flow (part 2)



system call/exception flow (part 2)



system call/exception flow (part 2)



trap-and-emulate (1)

normally: privileged instructions trigger fault

- e.g. accessing device memory directly (page fault)
- e.g. changing the exception table (protection fault)

normal OS: crash the program

hypervisor: pretend it did the right thing

- pretend kernel mode: the actual privileged operation
- pretend user mode: invoke guest's exception handler

trap-and-emulate (1)

normally: privileged instructions trigger fault

- e.g. accessing device memory directly (page fault)
- e.g. changing the exception table (protection fault)

normal OS: crash the program

hypervisor: pretend it did the right thing

- pretend kernel mode: the actual privileged operation
- pretend user mode: invoke guest's exception handler

trap-and-emulate: psuedocode

```
trap(...) {
  ...
  if (is_read_from_keyboard(tf->pc)) {
      do_read_system_call_based_on(tf);
  }
  ...
}
```

idea: translate privileged instructions into system-call-like operations

usually: need to deal with reading arguments, etc.

recall: xv6 keyboard I/O

```
...
data = inb(KBDATAP);
/* compiles to:
      mov $0x60, %edx
      in %dx, %al <-- FAULT IN USER MODE
 */
...
```

in user mode: triggers a fault

`in` instruction — read from special 'I/O address'

but same idea applies to mov from special memory address

more complete pseudocode (1)

```
trap(...) {  // tf = saved context (like xv6 trapframe)
  ...
  else if (exception_type == PROTECTION_FAULT
              && guest OS in kernel mode) {
    char *pc = tf->pc;
    if (is_in_instr(pc)) {  // interpret machine code!
      ...
      int src_address = get_instr_address(instrution);
      switch (src_address) {
        ...
        case KBDATAP:
          char c = do_syscall_to_read_keyboard();
          tf->registers[get_instr_dest(pc)] = c;
          tf->pc += get_instr_length(pc);
          break;
          ...
      }
    }
  }
```

more complete pseudocode (1)

```
trap(...) {  // tf = saved context (like xv6 trapframe)
  ...
  else if (exception_type == PROTECTION_FAULT
              && guest OS in kernel mode) {
    char *pc = tf->pc;
    if (is_in_instr(pc)) {  // interpret machine code!
      ...
      int src_address = get_instr_address(instrution);
      switch (src_address) {
        ...
        case KBDATAP:
          char c = do_syscall_to_read_keyboard();
          tf->registers[get_instr_dest(pc)] = c;
          tf->pc += get_instr_length(pc);
          break;
          ...
      }
    }
  }
```

more complete pseudocode (1)

```
trap(...) {  // tf = saved context (like xv6 trapframe)
  ...
  else if (exception_type == PROTECTION_FAULT
            && guest OS in kernel mode) {
    char *pc = tf->pc;
    if (is_in_instr(pc)) {  // interpret machine code!
      ...
      int src_address = get_instr_address(instrution);
      switch (src_address) {
        ...
        case KBDATAP:
          char c = do_syscall_to_read_keyboard();
          tf->registers[get_instr_dest(pc)] = c;
          tf->pc += get_instr_length(pc);
          break;
          ...
      }
    }
  }
```

trap-and-emulate (1)

normally: privileged instructions trigger fault

- e.g. accessing device memory directly (page fault)
- e.g. changing the exception table (protection fault)

normal OS: crash the program

hypervisor: pretend it did the right thing

- pretend kernel mode: the actual privileged operation
- pretend user mode: invoke guest's exception handler

more complete pseudocode (2)

```
trap(...) {   // tf = saved context (like xv6 trapframe)
  ...
  else if (exception_type == PROTECTION_FAULT
        && guest OS in user mode) {
    ...
    tf->in_kernel_mode = TRUE;
    tf->stack_pointer = /* guest OS kernel stack */;
    tf->pc = /* guest OS trap handler */;
  }
}
```

trap and emulate (2)

guest OS should still handle exceptions for its programs

most exceptions — just “reflect” them in the guest OS

look up exception handler, kernel stack pointer, etc.

saved by previous privilege instruction trap

reflecting exceptions

```
trap(...) {  ...
  else if ( exception_type == /* most exception type
        && guest OS in user mode) {
    ...
    tf->in_kernel_mode = TRUE;
    tf->stack_pointer = /* guest OS kernel stack */;
    tf->pc = /* guest OS trap handler */;
  }
```

trap and emulate (3)

what about memory mapped I/O?

when guest OS tries to access "magic" device address, get page fault

need to emulate any memory writing instruction!

trap and emulate (3)

what about memory mapped I/O?

when guest OS tries to access "magic" device address, get page fault

need to emulate any memory writing instruction!

(at least) two types of page faults for hypervisor

- guest OS trying to access device memory — emulate it
- guest OS trying to access memory not in *its* page table — run exception handler in guest

(and some more types — next topic)

trap and emulate not enough

trap and emulate assumption: can cause fault

priviliged instruction not in kernel

memory access not in hypervisor-set page table

…

until ISA extensions, on x86, not always possible

if time, (pretty hard-to-implement) workarounds later

things VM needs

normal user mode intructions

- just run it in user mode

guest OS I/O or other privileged instructions

- guest OS tries I/O/etc. — triggers interrupt
- hypervisor translates to I/O request
- or records privileged state change (e.g. switch to user mode) for later

guest OS exception handling

- track "guest OS thinks it in kernel mode"?
- record OS exception handler location when 'set handler' instruction faults
- hypervisor adjust PC, stack, etc. when guest OS should have exception

guest OS virtual memory

- ???

things VM needs

normal user mode intructions
- just run it in user mode

guest OS I/O or other privileged instructions
- guest OS tries I/O/etc. — triggers interrupt
- hypervisor translates to I/O request
- or records privileged state change (e.g. switch to user mode) for later

guest OS exception handling
- track "guest OS thinks it in kernel mode"?
- record OS exception handler location when 'set handler' instruction faults
- hypervisor adjust PC, stack, etc. when guest OS should have exception

guest OS virtual memory
- ???

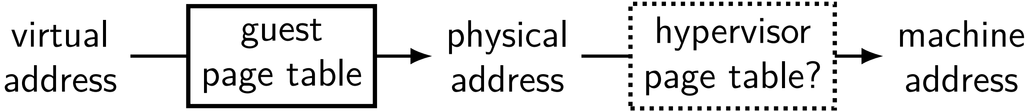
terms for this lecture

virtual address — virtual address for guest OS

physical address — physical address for guest OS

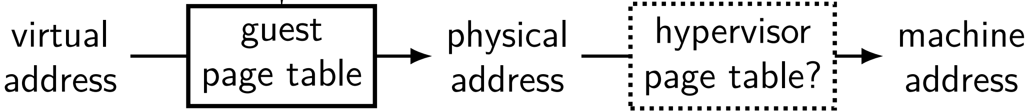
machine address — physical address for hypervisor/host OS

three page tables

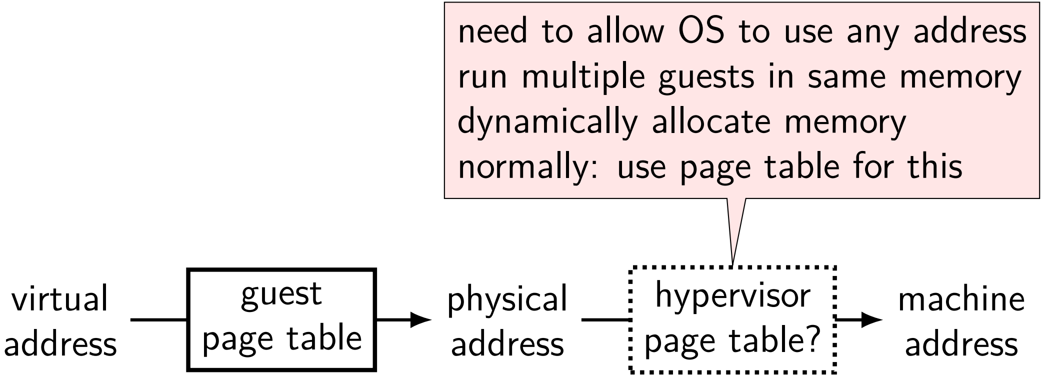


three page tables

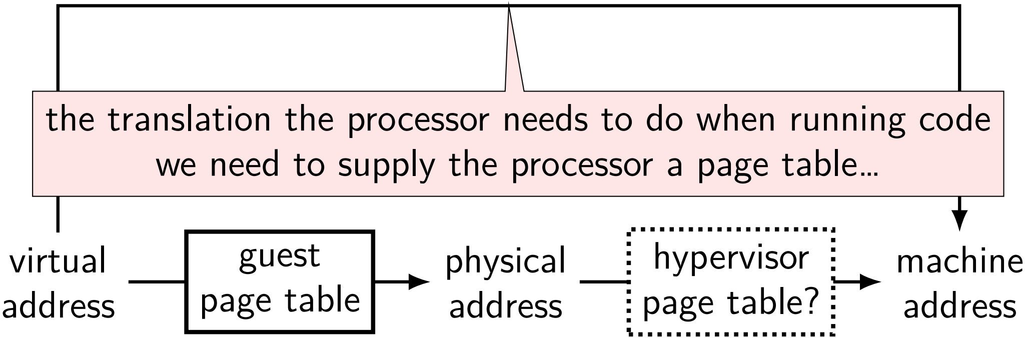
page table pointer guest
set with privileged instruction
(x86: `mov ..., %cr3`)
hypervisor records on protection fault



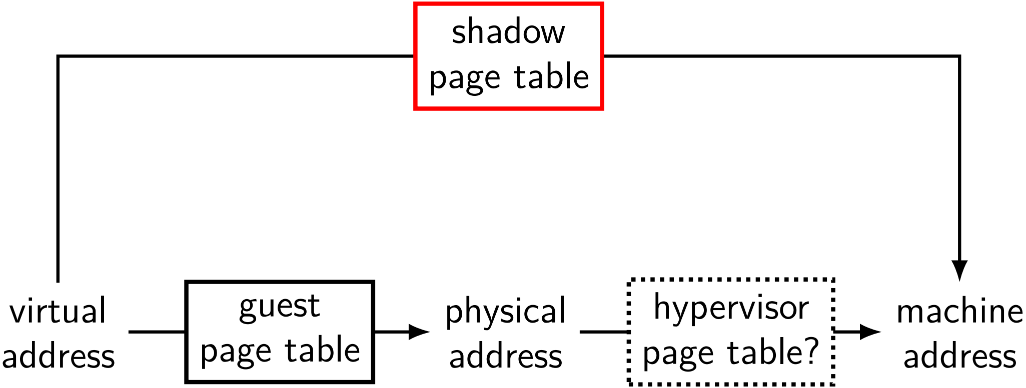
three page tables



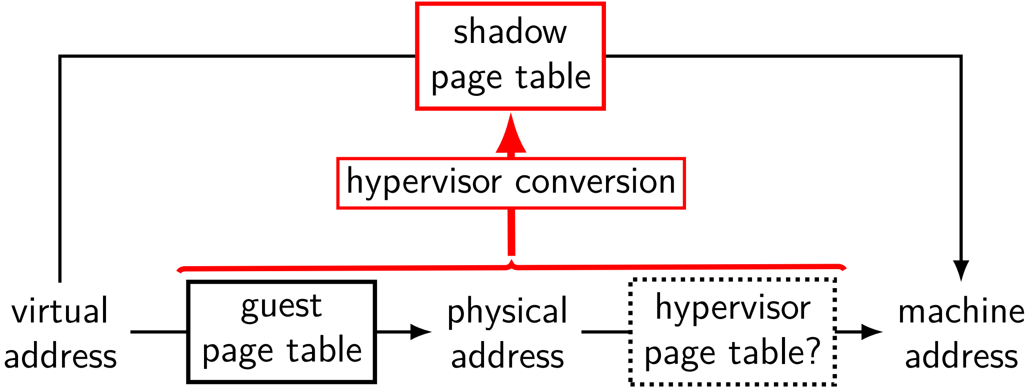
three page tables



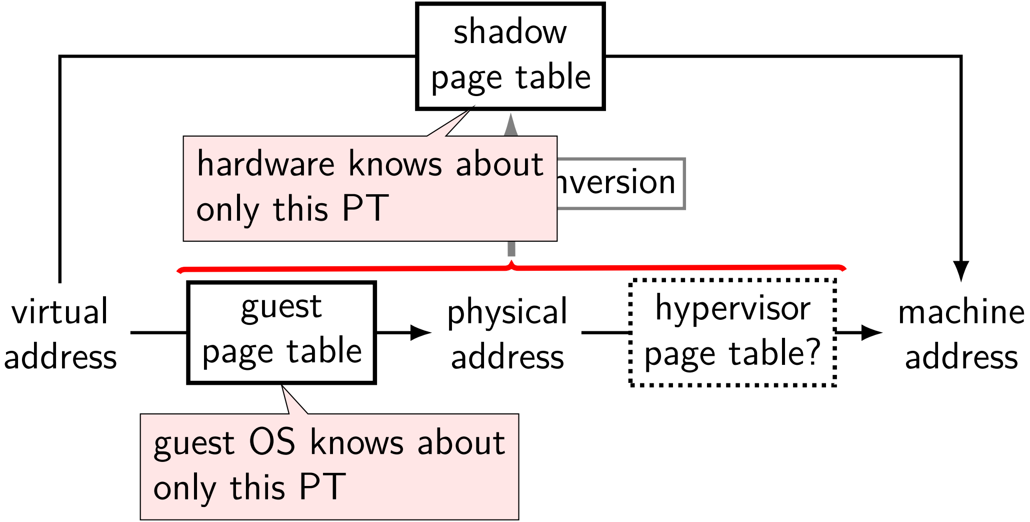
three page tables



three page tables



three page tables



page table synthesis question

creating new page table = two PT lookups

- lookup in guest OS page table
- lookup in hypervisor page table (or equivalent)

synthesize new page table from combined info

page table synthesis question

creating new page table = two PT lookups

- lookup in guest OS page table
- lookup in hypervisor page table (or equivalent)

synthesize new page table from combined info

Q: when does the hypervisor update the shadow page table?

interlude: the TLB

Translation **L**ookaside **B**uffer — cache for page table entries

what the processor actually uses to do address translation with normal page tables

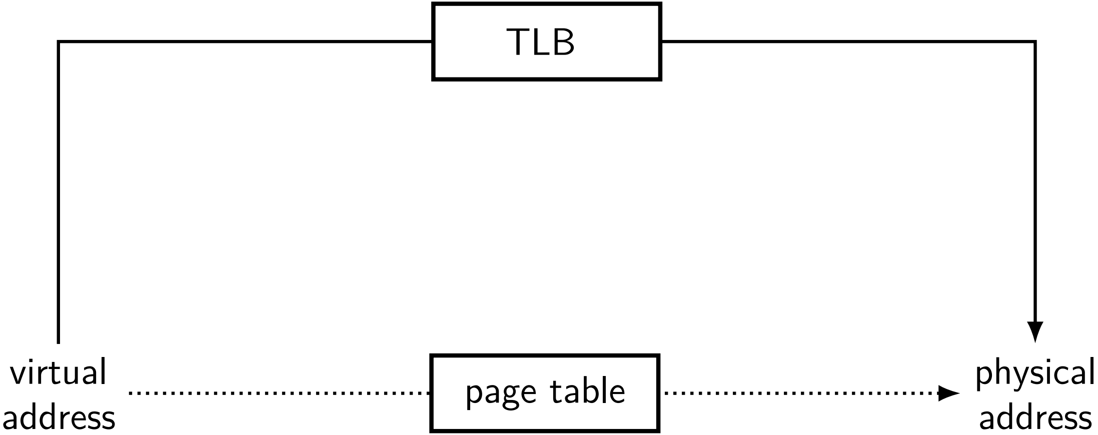
has the same problem

contents synthesized from the 'normal' page table

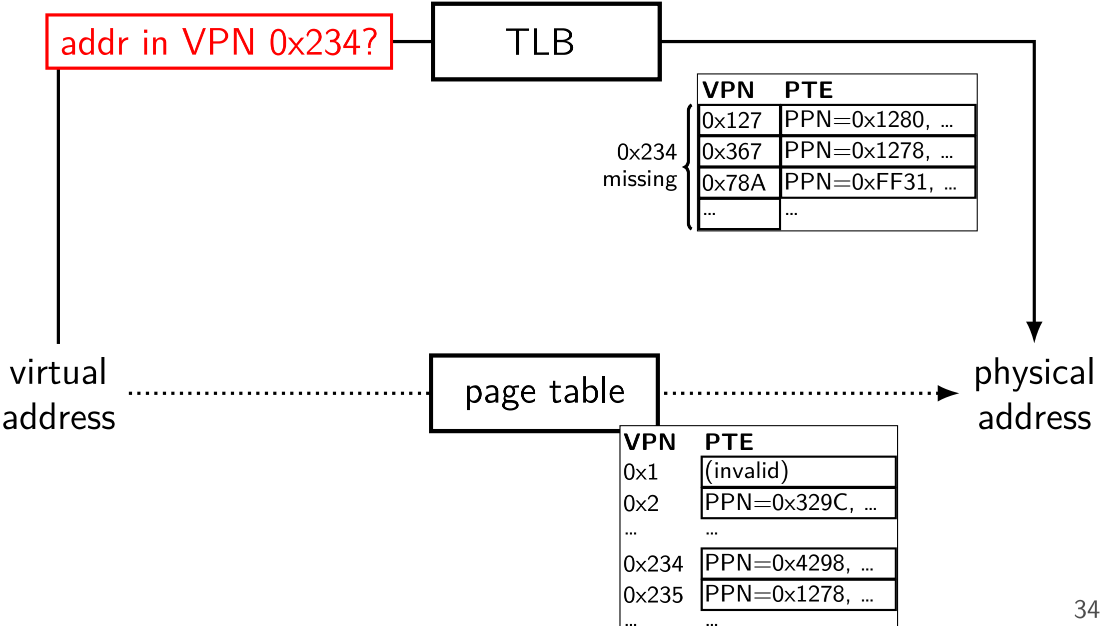
processor needs to decide when to update it

preview: hypervisor can use same solution

Interlude: TLB (no virtualization)



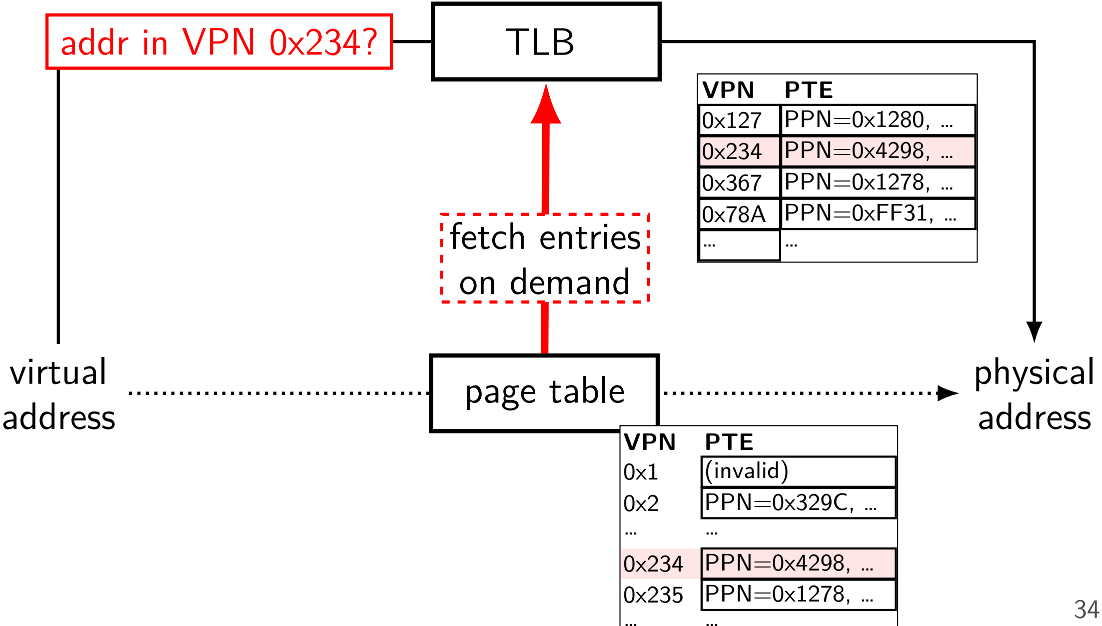
Interlude: TLB (no virtualization)



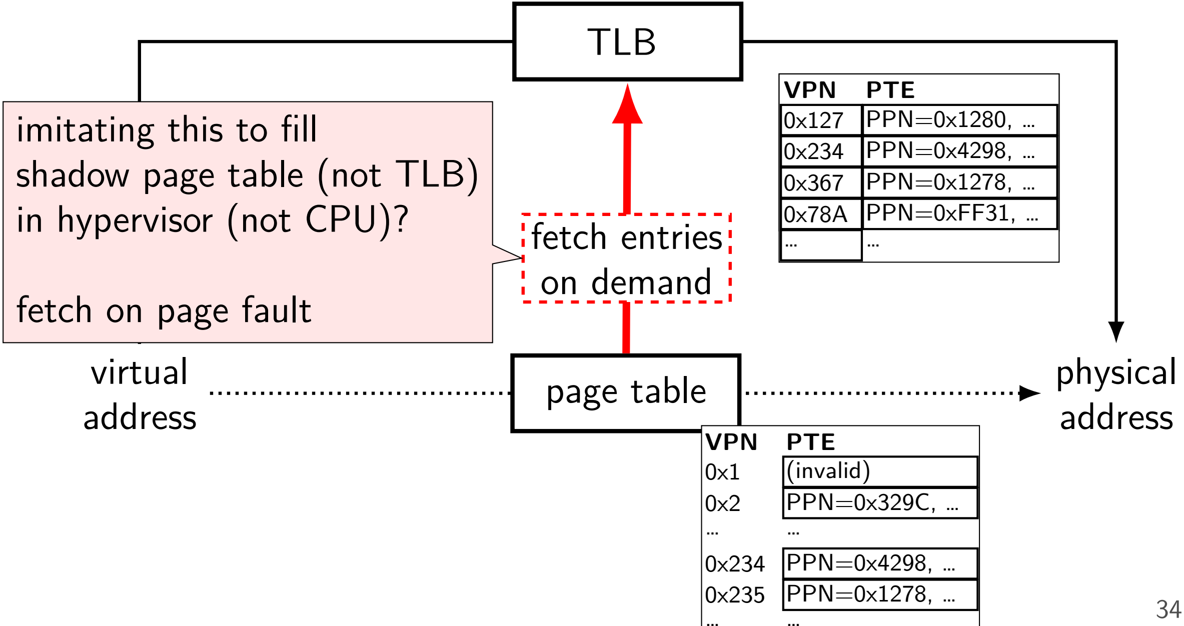
VPN	PTE
0x127	PPN=0x1280, ...
0x367	PPN=0x1278, ...
0x78A	PPN=0xFF31, ...
...	...

VPN	PTE
0x1	(invalid)
0x2	PPN=0x329C, ...
...	...
0x234	PPN=0x4298, ...
0x235	PPN=0x1278, ...
...	...

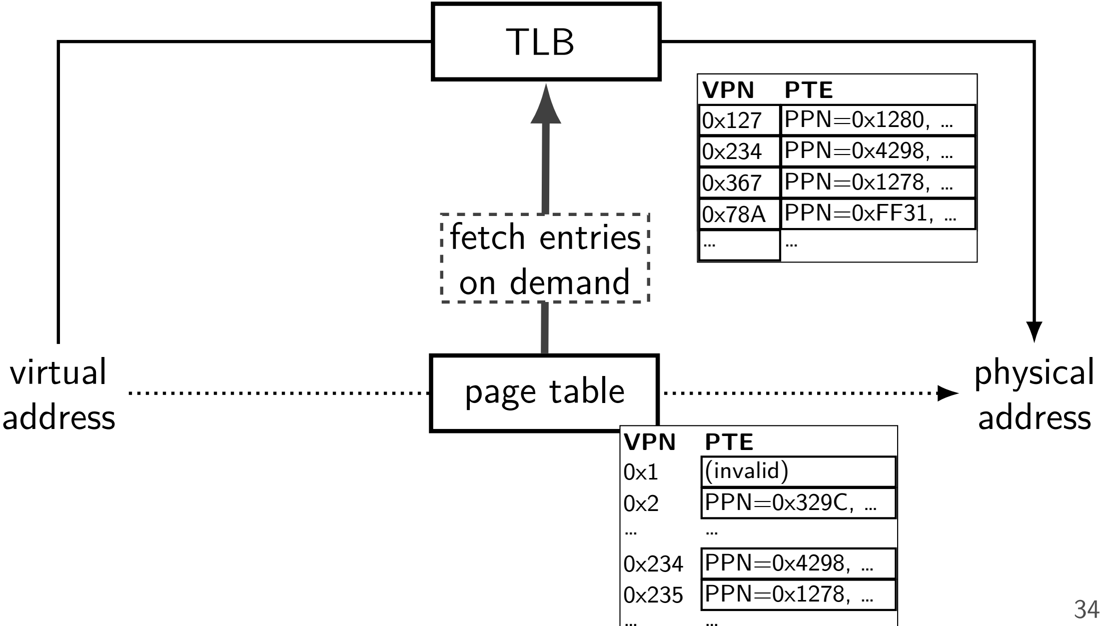
Interlude: TLB (no virtualization)



Interlude: TLB (no virtualization)



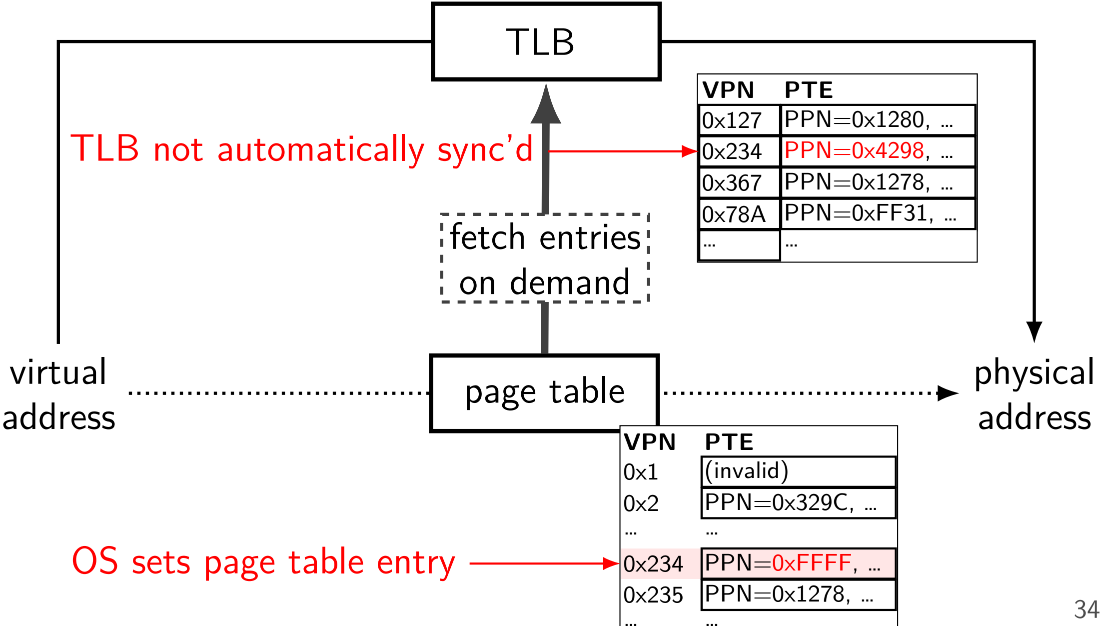
Interlude: TLB (no virtualization)



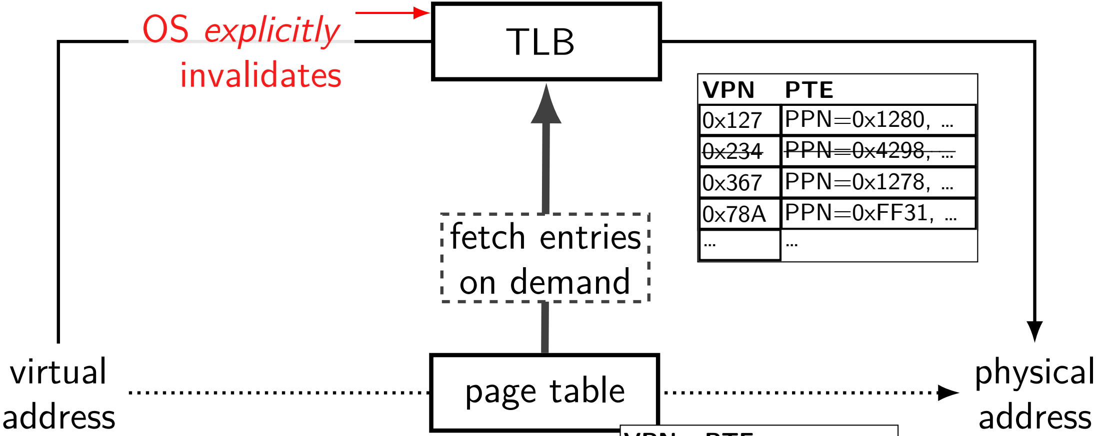
VPN	PTE
0x127	PPN=0x1280, ...
0x234	PPN=0x4298, ...
0x367	PPN=0x1278, ...
0x78A	PPN=0xFF31, ...
...	...

VPN	PTE
0x1	(invalid)
0x2	PPN=0x329C, ...
...	...
0x234	PPN=0x4298, ...
0x235	PPN=0x1278, ...
...	...

Interlude: TLB (no virtualization)



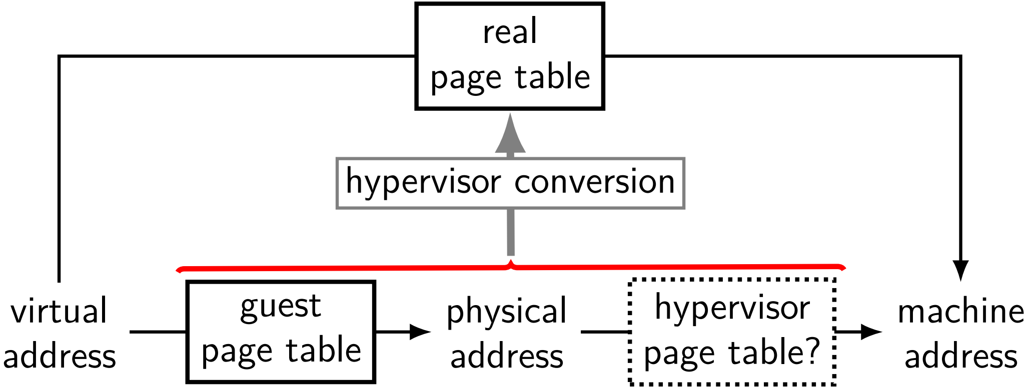
Interlude: TLB (no virtualization)



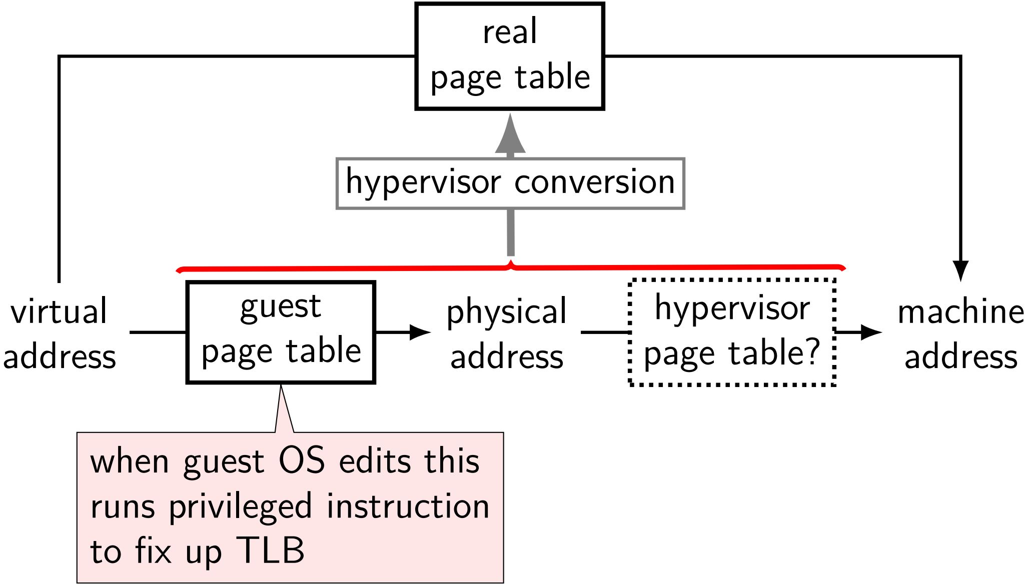
VPN	PTE
0x127	PPN=0x1280, ...
~~0x234~~	~~PPN=0x4298, ...~~
0x367	PPN=0x1278, ...
0x78A	PPN=0xFF31, ...
...	...

VPN	PTE
0x1	(invalid)
0x2	PPN=0x329C, ...
...	...
0x234	PPN=0xFFFF, ...
0x235	PPN=0x1278, ...
...	...

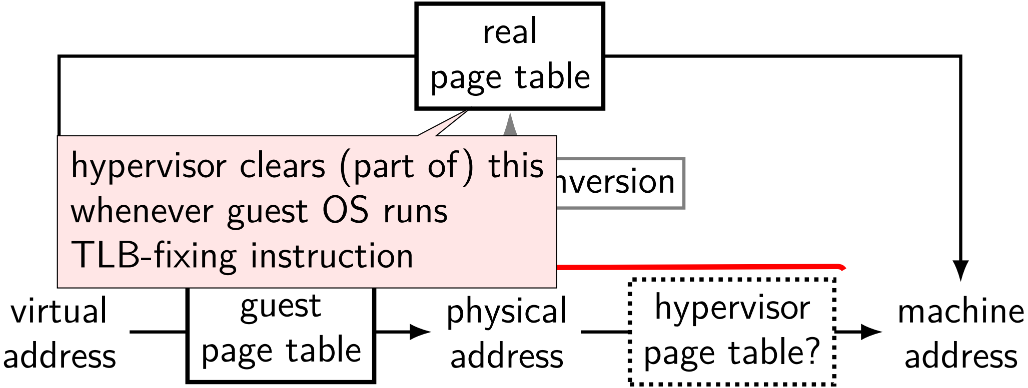
three page tables (revisited)



three page tables (revisited)



three page tables (revisited)



alternate view of shadow page table

shadow page table is like a *virtual TLB*

caches commonly used page table entries in guest

entries need to be in shadow page table for instructions to run

needs to be explicitly cleared by guest OS

implicitly filled by hypervisor

on TLB invalidation

two major ways to invalidate TLB:

when setting a new page table base pointer

e.g. x86: `mov ..., %cr3`

when running an explicit invalidation instruction

e.g. x86: `invlpg`

hopefully, both privileged instructions

nit: memory-mapped I/O

recall: devices which act as 'magic memory'

hypervisor needs to emulation

keep corresponding pages invalid for trap+emulate

page fault triggers instruction emulation instead

problem with filling on demand

most OSs: invalidate *entire TLB* on context switch

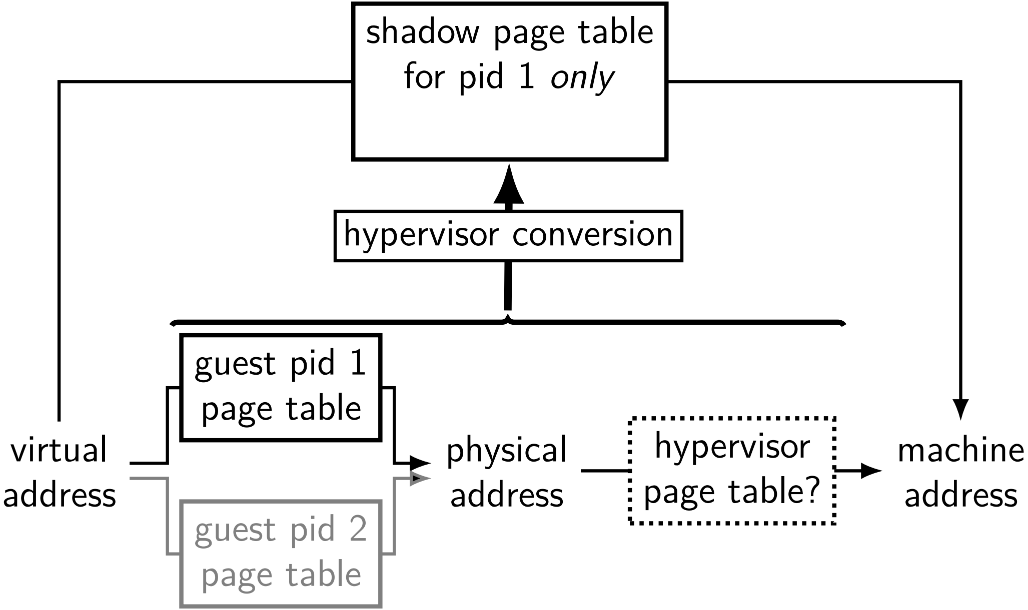
so, rebuild shadow page table on each guest OS context switch

this is often unacceptably slow

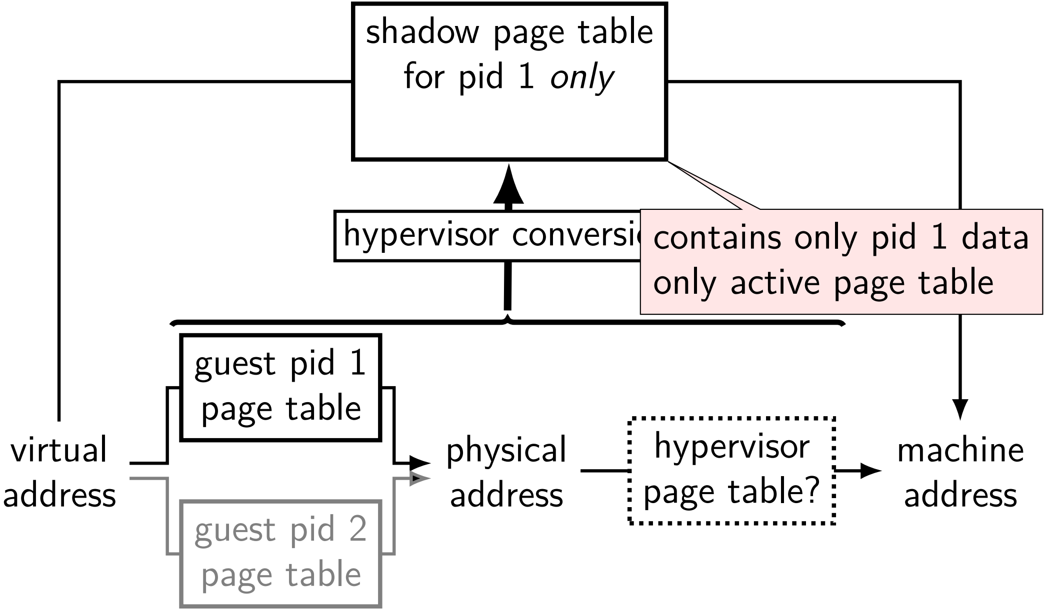
want to cache the shadow page tables

problem: OS won't tell you when it's writing

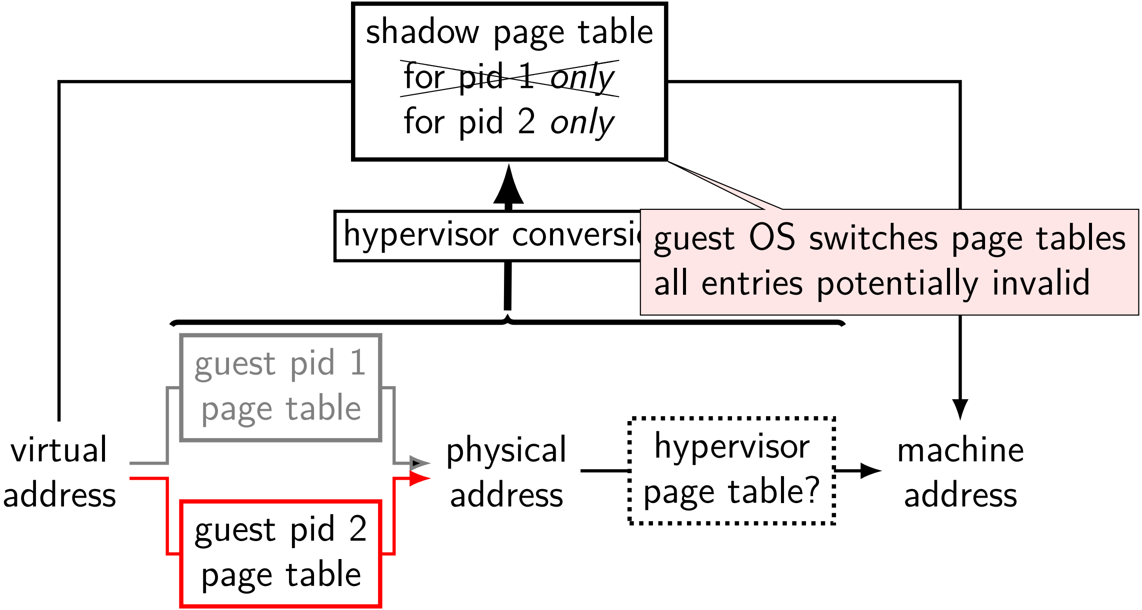
problem with filling on demand



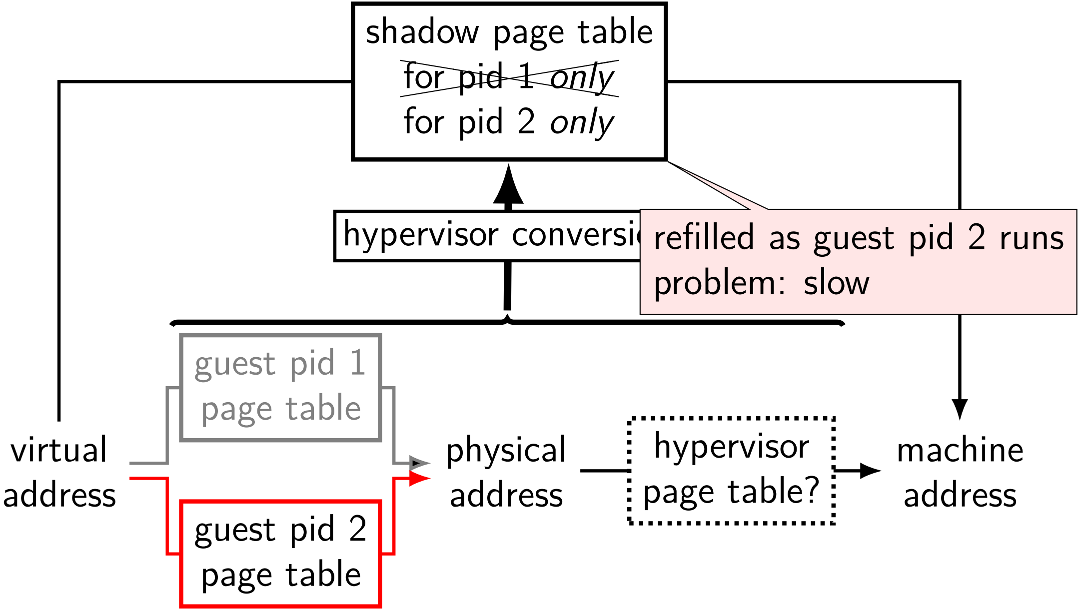
problem with filling on demand



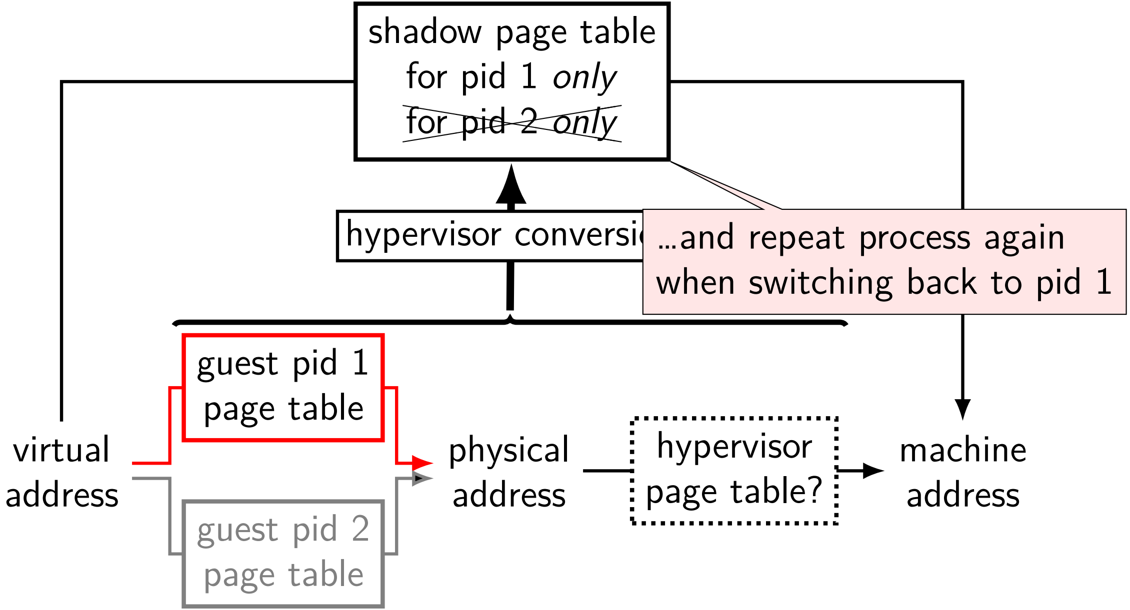
problem with filling on demand



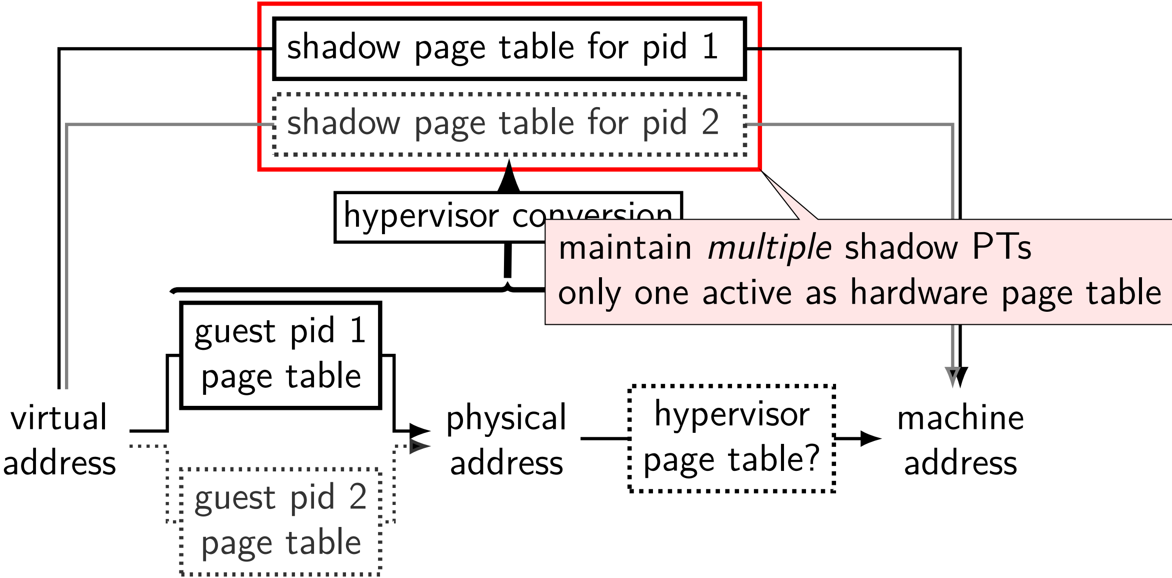
problem with filling on demand



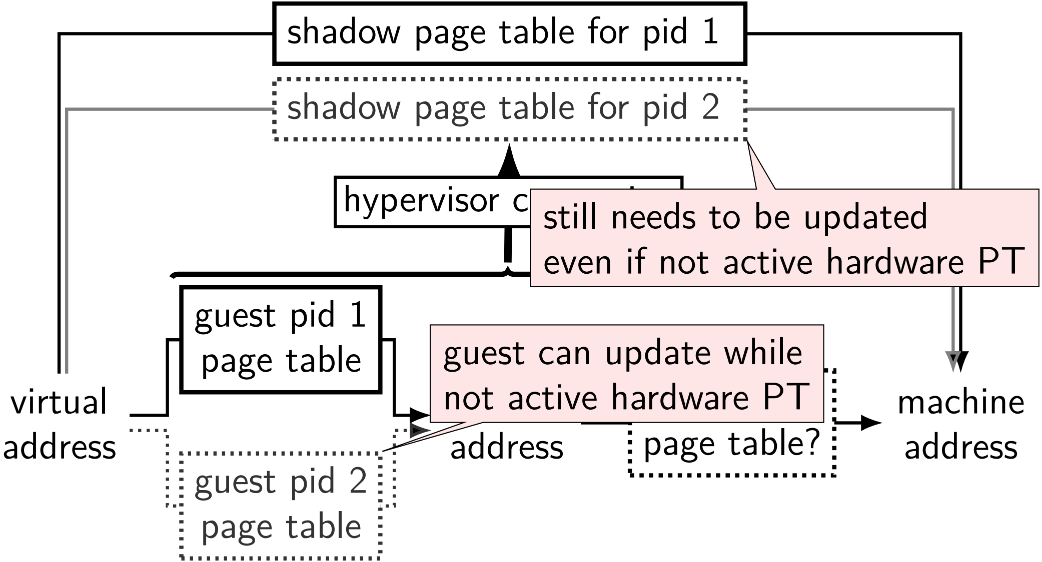
problem with filling on demand



proactively maintaining page tables



proactively maintaining page tables



proactively maintaining page tables

track physical pages that are part of any page tables

- update list on page table base register write?
- update list while filling shadow page table on demand

make sure marked read-only in shadow page tables

use trap+emulate to handles writes to them

(...even if not current active guest page tables)

on write to page table: update shadow page table

pros/cons: proactive over on-demand

pro: work with guest OSs that make assumptions about TLB size

pro: maintain shadow page table for each guest process

can avoid reconstructing each page table on each context switch

con: more instructions spent doing copy-on-write

con: what happens when page table memory recycled?

page tables and kernel mode?

guest OS can have *kernel-only* pages

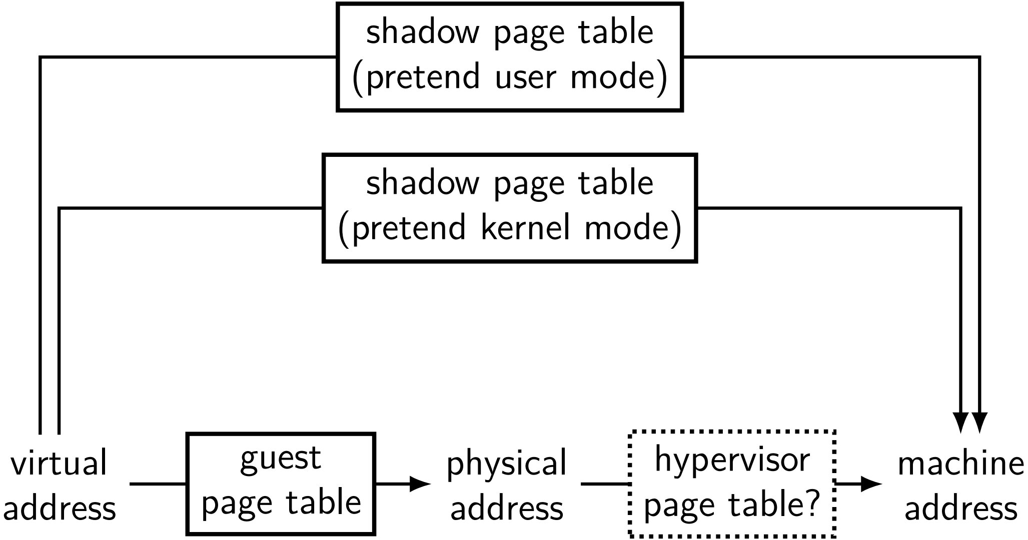
guest OS in pretend kernel mode

- shadow PTE: marked as user-mode accessible

guest OS in pretend user mode

- shadow PTE: marked inaccessible

four page tables? (1)



four page tables? (2)

one solution: pretend kernel and pretend user shadow page table

alternative: clear page table on kernel/user switch

neither seems great for overhead

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?

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

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

cache converted code

- `translate_and_run` checks cache first

patch calls to `translate_and_run` to refer directly 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

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 normal

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

HW support 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

nested page tables

- HW does lookup in guest page table, then hypervisor PT
- avoids extra page faults

tagging TLB entries with the VM ID

- keep page table entries cached despite switching from guest to hypervisor PT