virtual machines

Changelog

Changes made in this version not seen in first lecture:

- 23 April 2019: rearrange slide order to better match lecture order
- 23 April 2019: change 'real page table' to 'shadow page table' in some places
- 23 April 2019: move layering slide earlier

capabilities

token to identify = permission to access

typically *opaque* token

some capability list examples

file descriptors

list of open files process has acces to

page table (sort of?)

list of physical pages process is allowed to access

some capability list examples

file descriptors

list of open files process has acces to

page table (sort of?)

list of physical pages process is allowed to access

list of what process can access *stored with process*

handle to access object = key in permitted object table

impossible to skip permission check!

sharing capabilities

capability-based OSes have ways of sharing capabilities:

inherited by spawned programs

- file descriptors/page tables do this

send over local socket or pipe

- usually supported for file descriptors!
- (look up `SCM_RIGHTS` — how it works different for Linux v. OS X v. FreeBSD v. …)

Capsicum: practical capabilities for UNIX (1)

Capsicum: research project from Cambridge

adds capabilities to FreeBSD by extending file descriptors

opt-in: can set process to require capabilities to access objects

- instead of absolute path, process ID, etc.

capabilities = fds for each directory/file/process/etc.

more permissions on fds than read/write

- execute
- open files in (for fd representing directory)
- kill (for fd reporesenting process)
- ...

Capsicum: practical capabilities for UNIX (2)

capabilities = no global names

no filenames, instead fds for directories

- new syscall: `openat(directory_fd, "path/in/directory")`
- new syscall: `fexecv(file_fd, argv)`

no pids, instead fds for processes

- new syscall: `pdfork()`

alternative to per-process tables

file descriptors: different in every process

- use special functions to move between processes

alternate idea: same number in every process

- one big table

sharing token = copy number

but how to control access? make numbers hard to guess

example: use random 128-bit numbers

sandboxing

sandbox — restricted environment for program

idea: dangerous code can play in the sandbox as much as it wants

can't do anything harmful

sandbox use cases

buggy video parsing code that has buffer overflows

browser running scripts in webpage

autograder running student submissions

…

sandbox use cases

buggy video parsing code that has buffer overflows

browser running scripts in webpage

autograder running student submissions

...

(parts of) program that don't need to have user's full permissions

- no reason video parsing code should be able open() my taxes

can we have a way to ask OS for this?

Google Chrome architecture

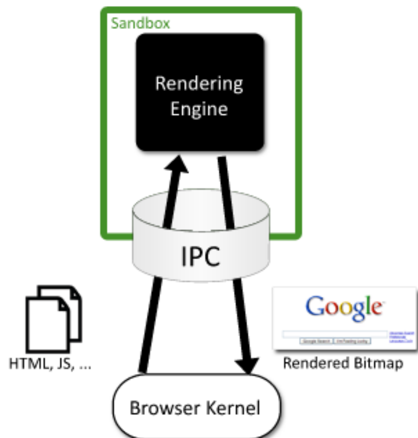


Figure 1: The browser kernel treats the rendering engine as a black box that parses web content and emits bitmaps of the rendered document.

sandboxing mechanisms

create a new user with few privileged, switch to user

- problem: creating new users usually requires sysadmin access
- problem: every user can do too much
- e.g. everyone can open network connection?

with capabilities, just discard most capabilities

- just close capabilities you don't need
- run rendering engine with only pipes to talk to browser kernel

otherwise: system call filtering

- disallow all 'dangerous' system calls

Linux system call filtering

`seccomp()` system call

"strict mode": only allow read/write/_exit/sigreturn

- current thread gives up all other privileges
- usage: setup pipes, then communicate with rest of process via pipes

alternately: setting a whitelist of allowed system calls + arguments

- little programming language (!) for supported operations

browsers use this to protect from bugs in their scripting implementations

- hope: find a way to execute arbitrary code? — not actually useful

sandbox browser setup

create pipe

spawn subprocess (“rendering engine”)

put subproces in strict system call filter mode

send subprocesses webpages + events

subprocess sends images to render back on pipe

sandboxing use case: buggy video decoder

```
/* dangerous video decoder to isolate */
int main() {
    EnterSandbox();
    while (fread(videoData, sizeof(videoData), 1, stdin) > 0) {
        doDangerousVideoDecoding(videoData, imageData);
        fwrite(imageData, sizeof(imageData), 1, stdout);
    }
}

/* code that uses it */
    FILE *fh = RunProgramAndGetFileHandle("./video-decoder");
    for (;;) {
        fwrite(getNextVideoData(), SIZE, 1, fh);
        fread(image, sizeof(image), 1, fh);
        displayImage(image);
    }
```

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

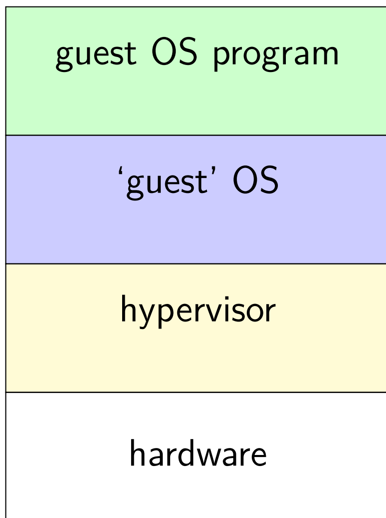
...or might not have time to mention

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

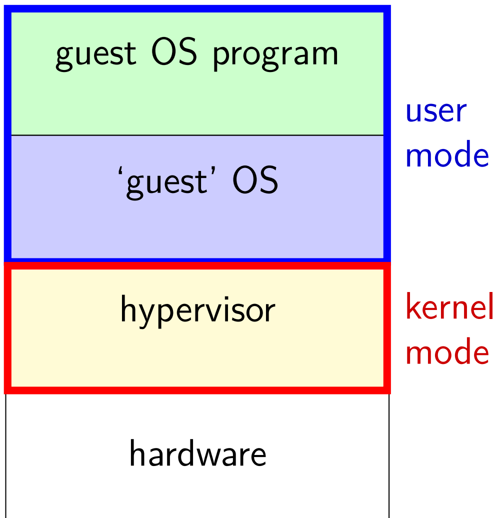
one variation: compile guest OS code to new machine code

- not as slow as you'd think, sometimes

VM layering (intro)



VM layering (intro)

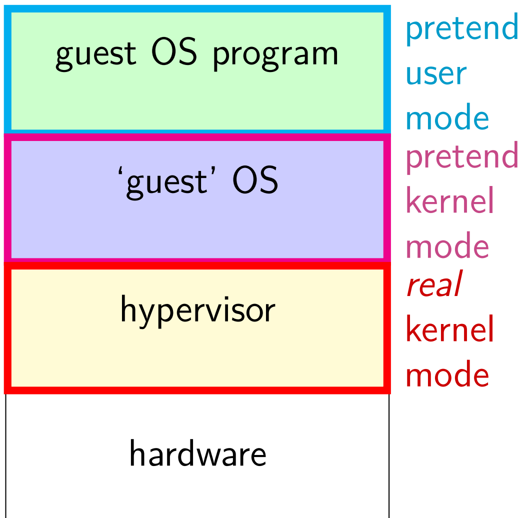


user mode

kernel mode

≈ hypervisor's process

VM layering (intro)

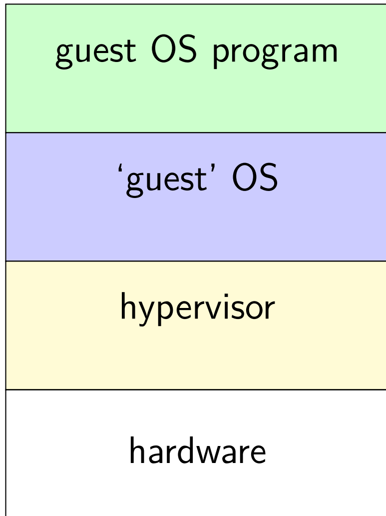


pretend user mode

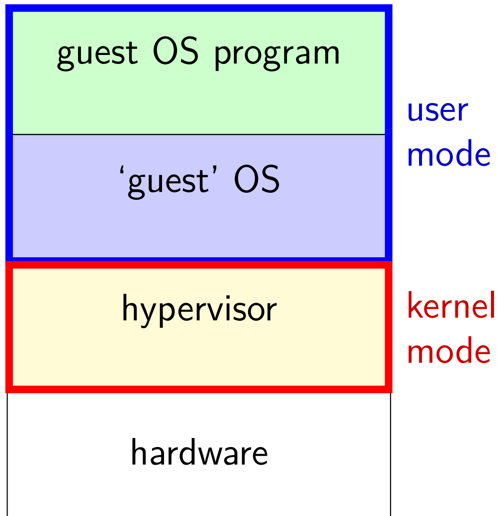
pretend kernel mode

real kernel mode

VM layering



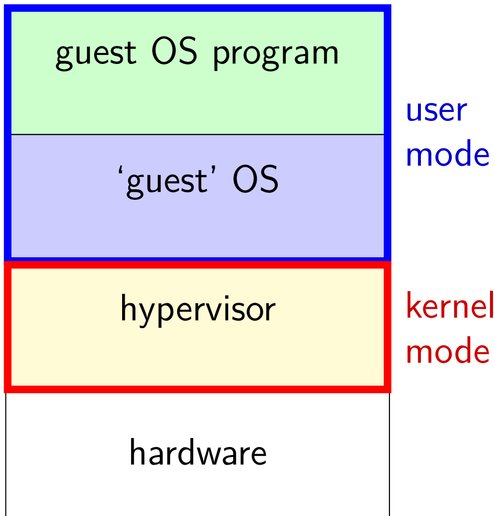
VM layering



hypervisor tracks...

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

VM layering

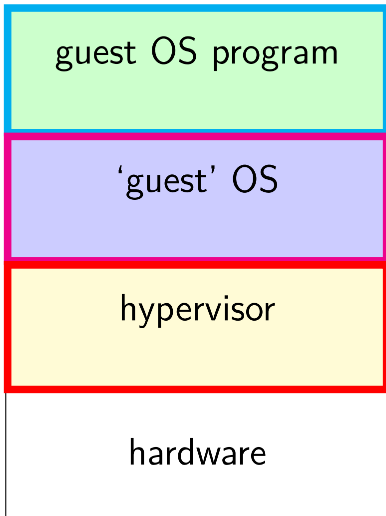


hypervisor tracks...

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



pretend user mode

pretend kernel mode

real kernel mode

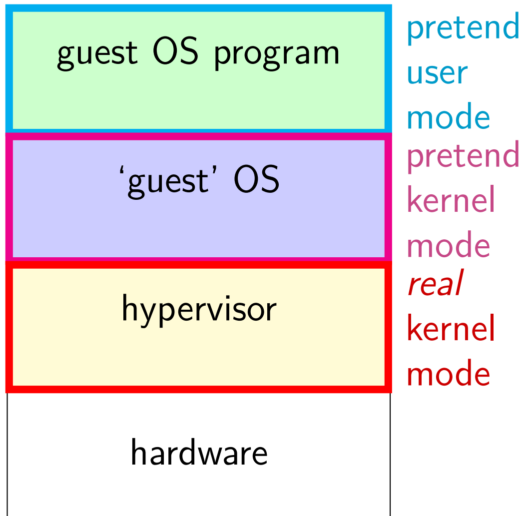
hypervisor tracks...

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



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

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

trap-and-emulate (1)

normally: privileged instructions trigger fault

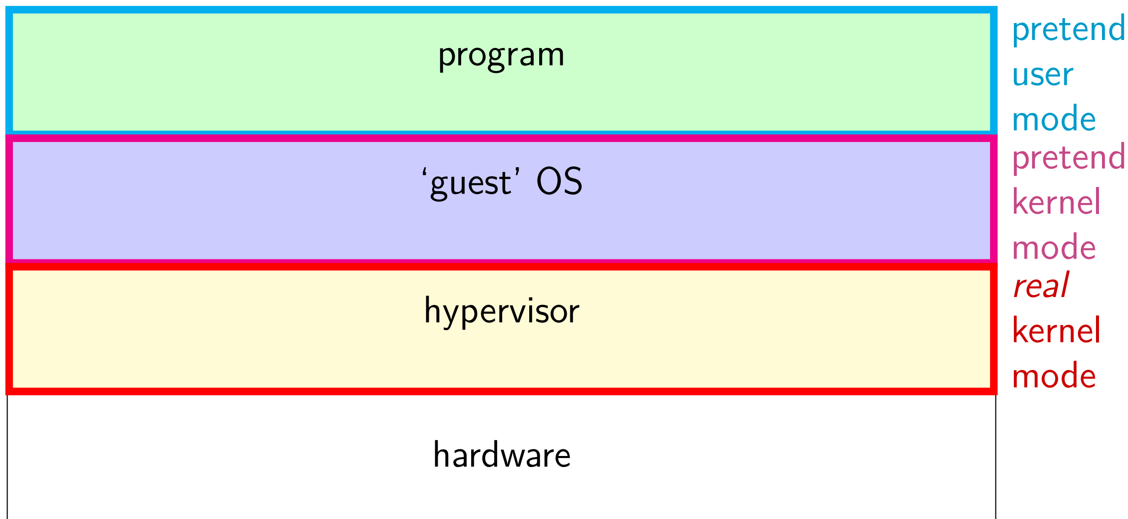
- e.g. accessing device memory directly (page fault)
- e.g. changing the exception table (protection fault)

normal OS: crash the program

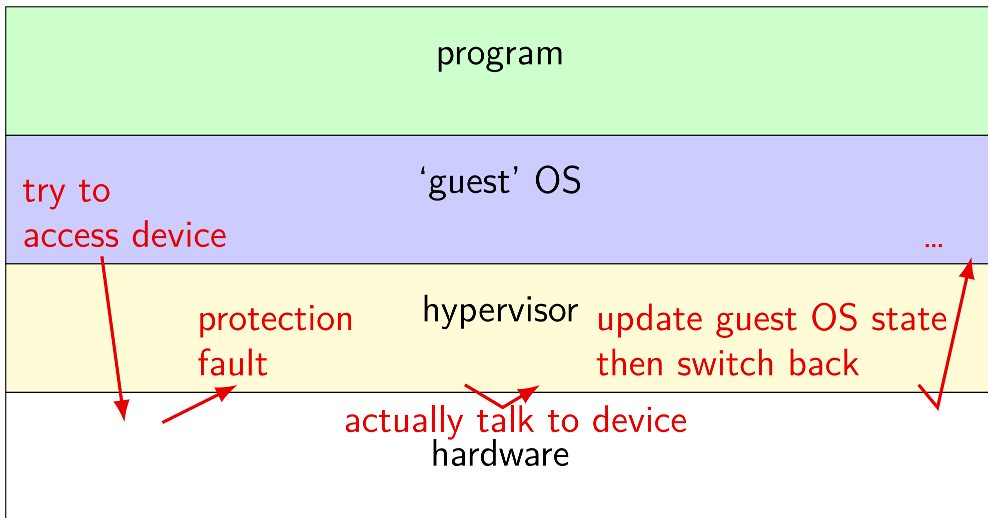
hypervisor: pretend it did the right thing

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

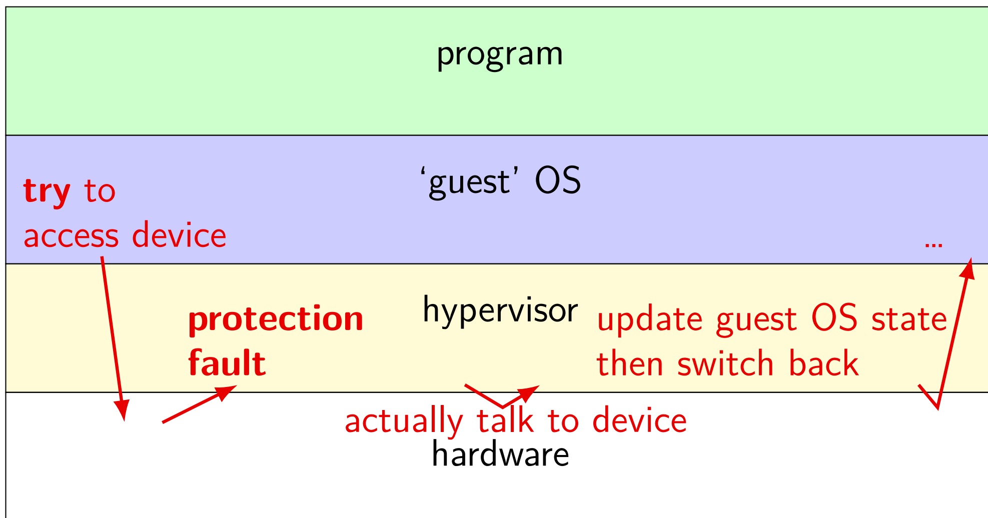
privileged I/O flow



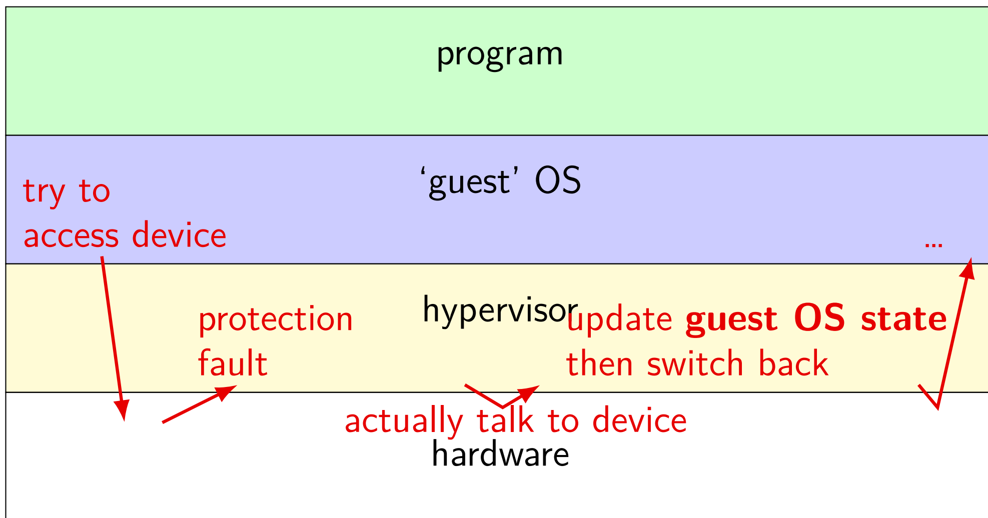
privileged I/O flow



privileged I/O flow



privileged I/O flow



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 + page fault

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

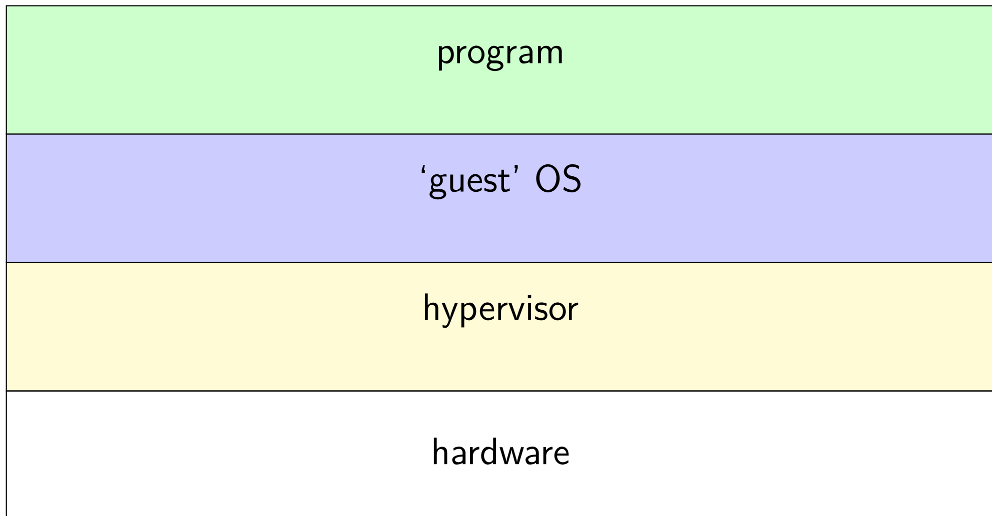
hypervisor: pretend it did the right thing

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

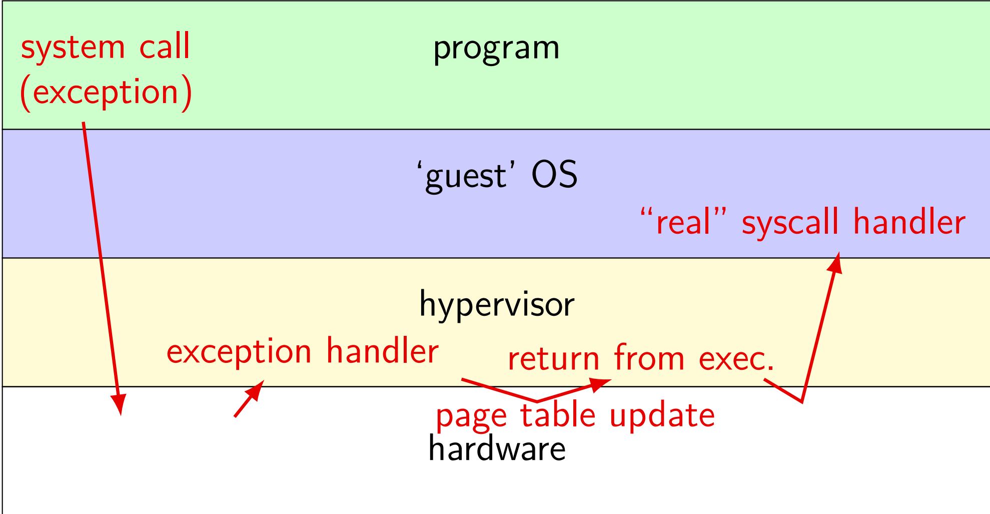
more complete pseudocode (2)

```
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 */;
  }
}
```

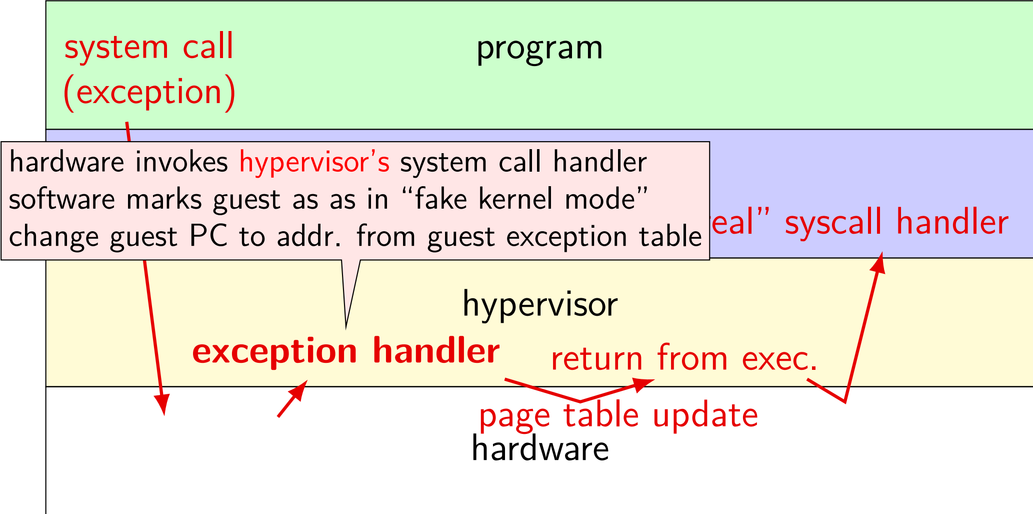
system call/exception flow (part 1)



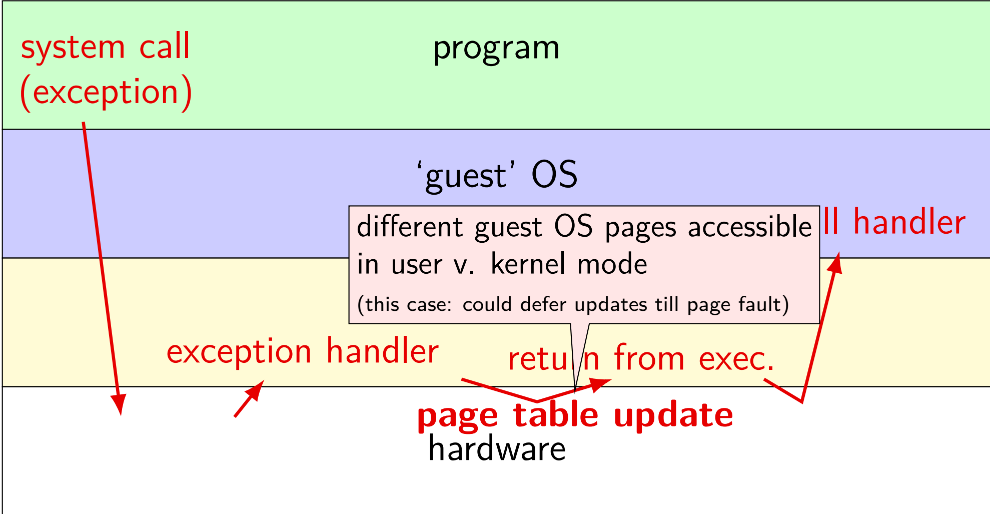
system call/exception flow (part 1)



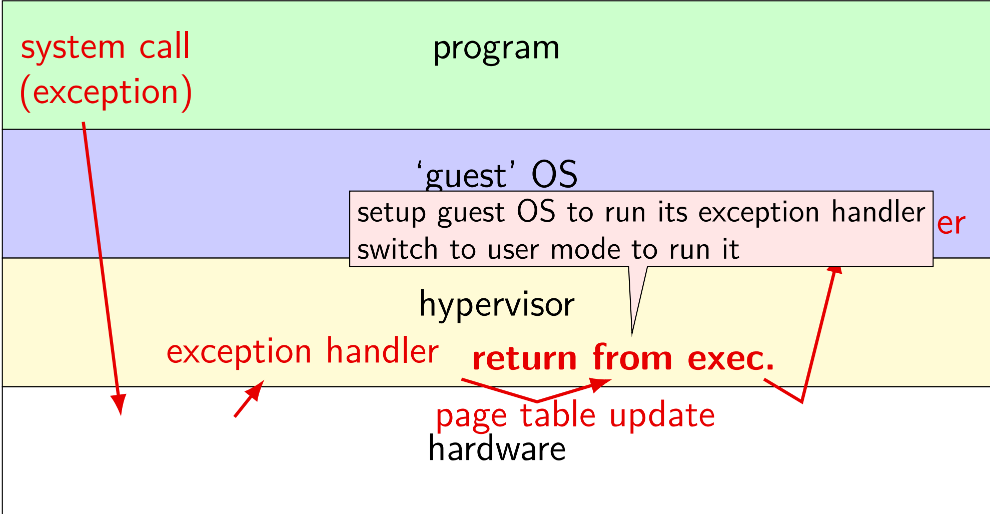
system call/exception flow (part 1)



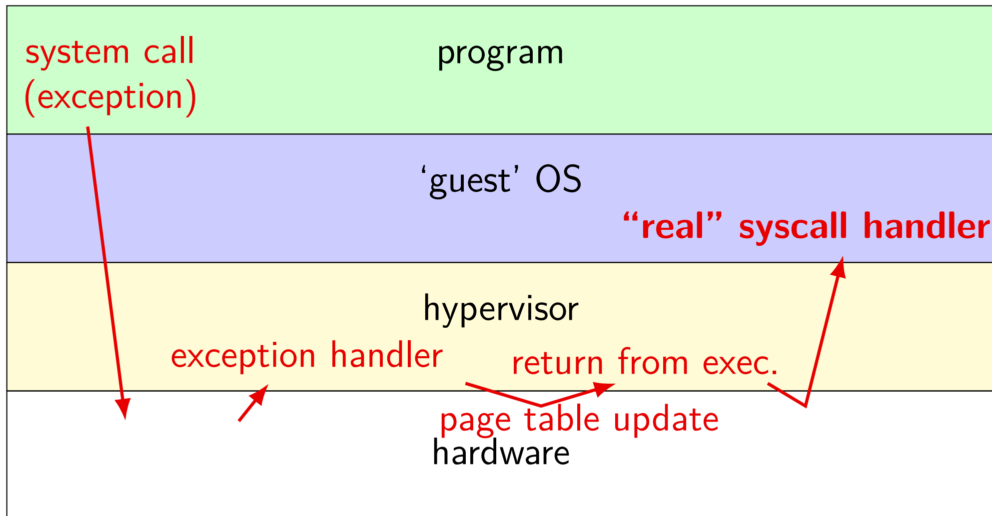
system call/exception flow (part 1)



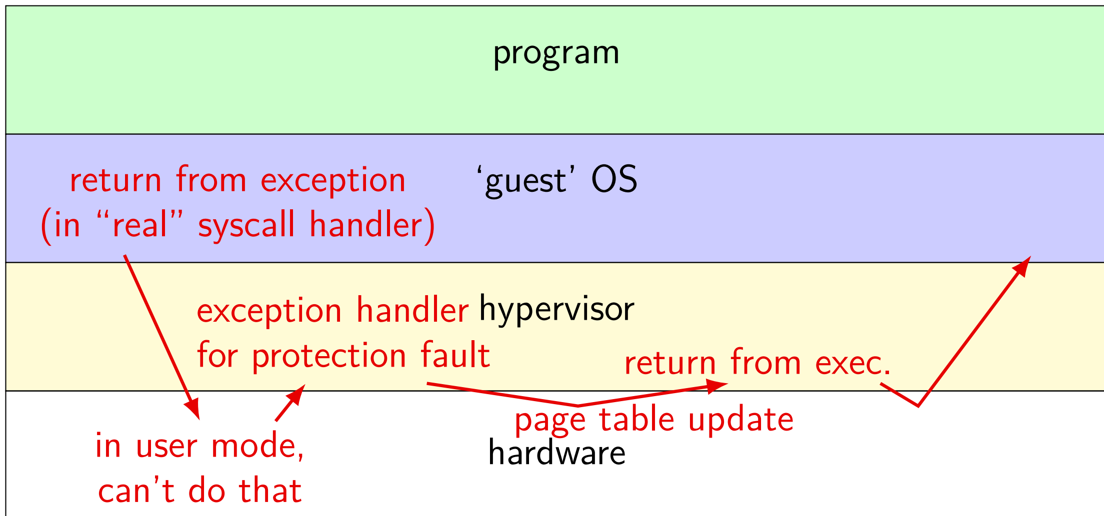
system call/exception flow (part 1)



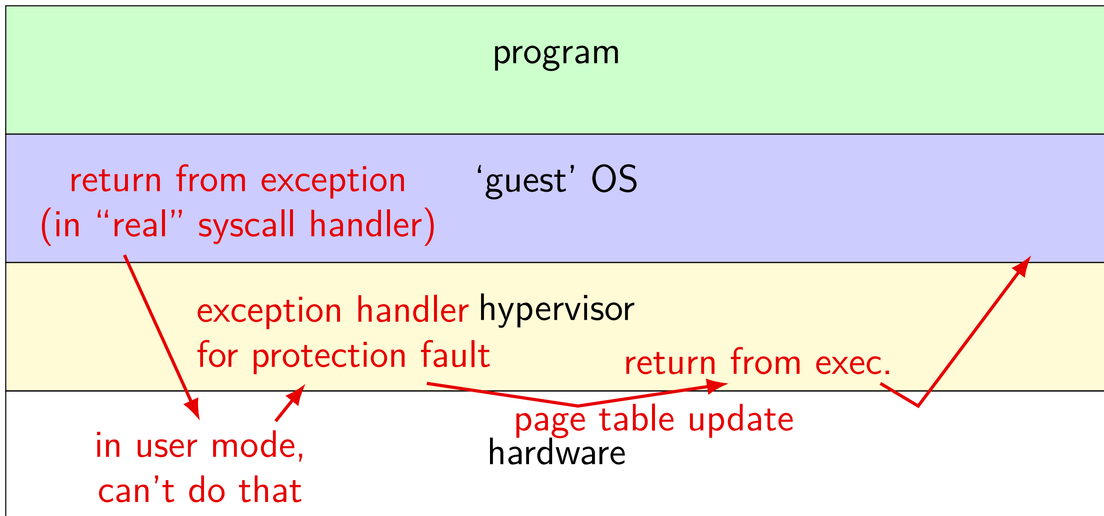
system call/exception flow (part 1)



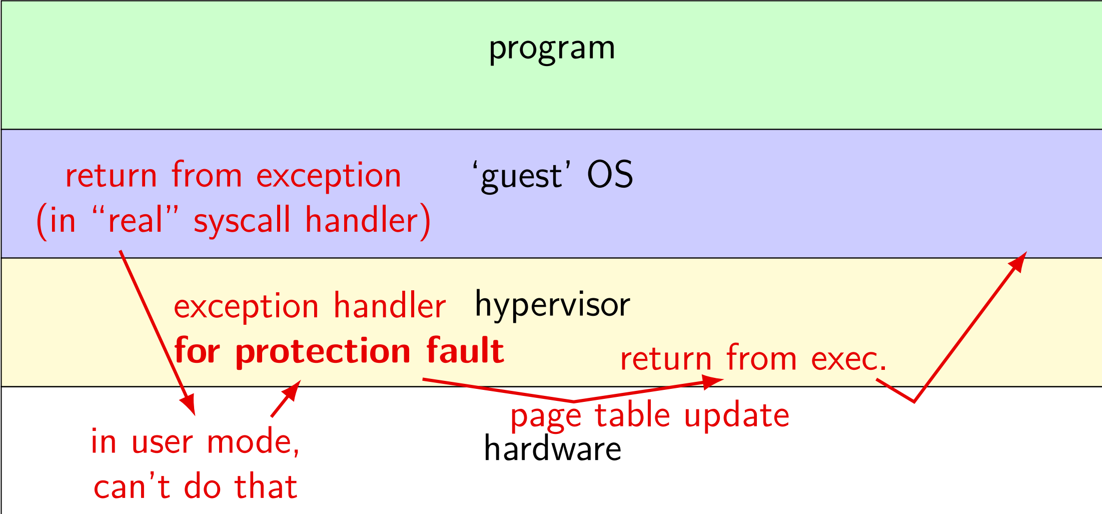
system call/exception flow (part 2)



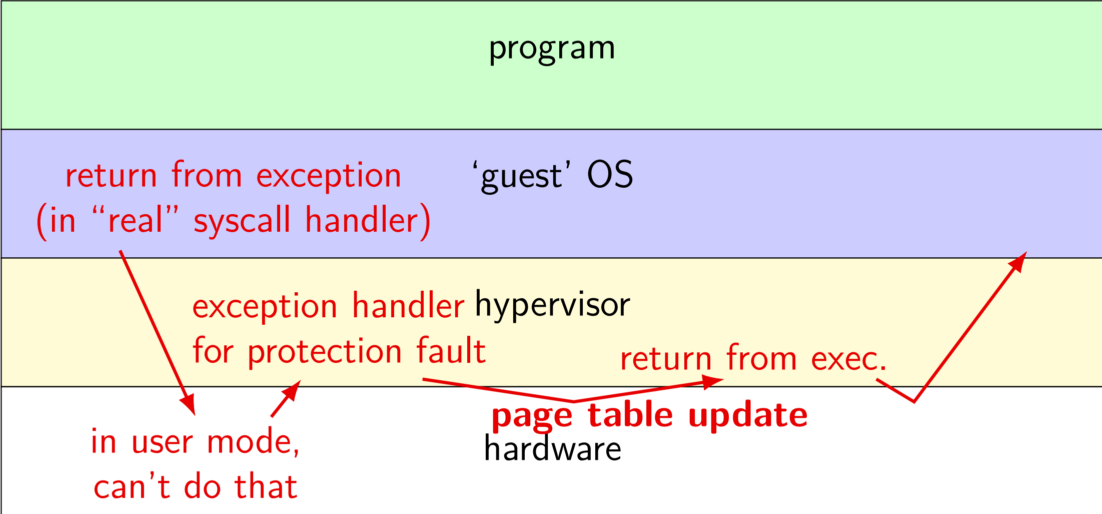
system call/exception flow (part 2)



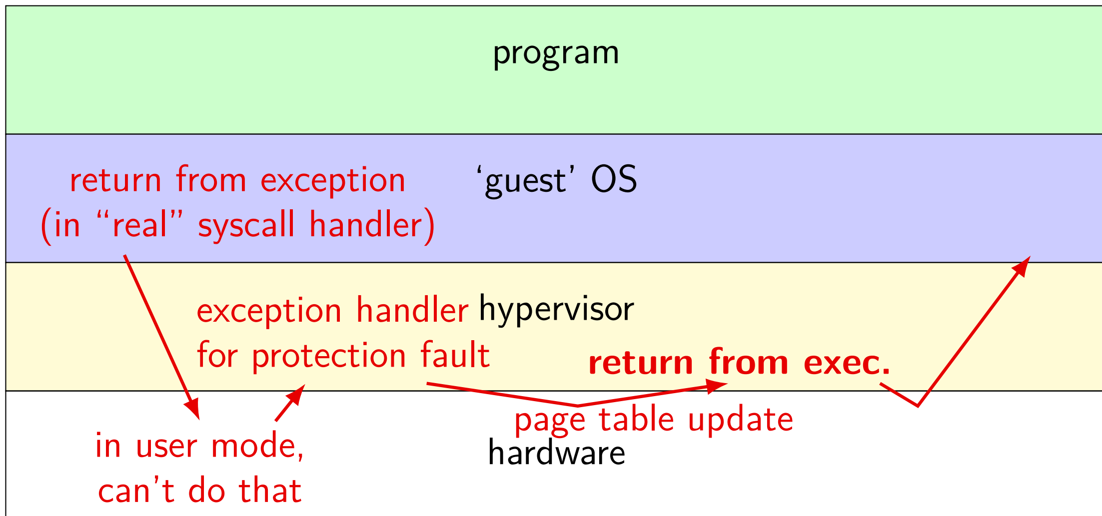
system call/exception flow (part 2)



system call/exception flow (part 2)



system call/exception flow (part 2)



trap and emulate (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 types */
        && 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 exception
- 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 exception
- 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

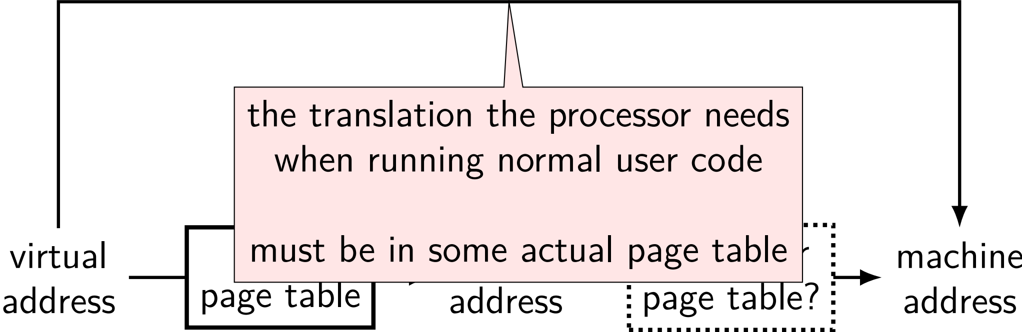
three page tables

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

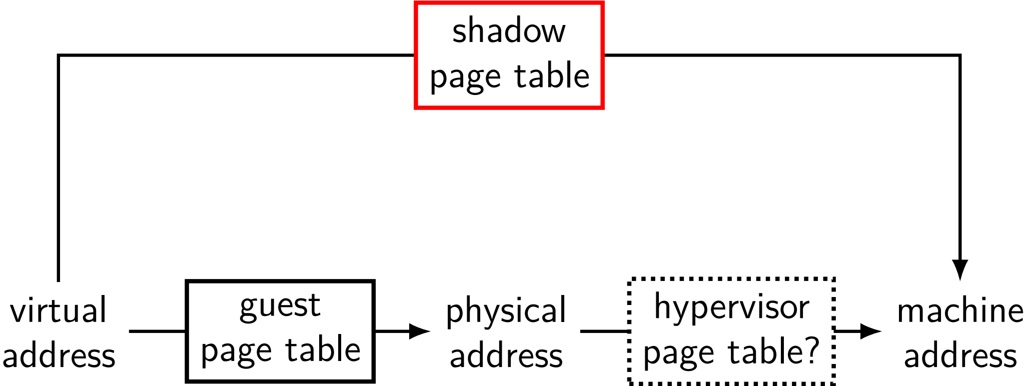
three page tables

need to allow OS to use any address
run multiple guests in same memory
dynamically allocate memory
normally: use page table for this

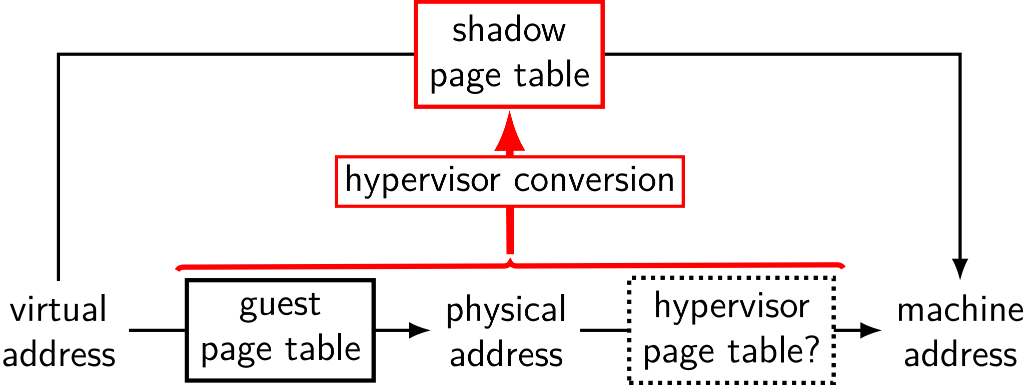
three page tables



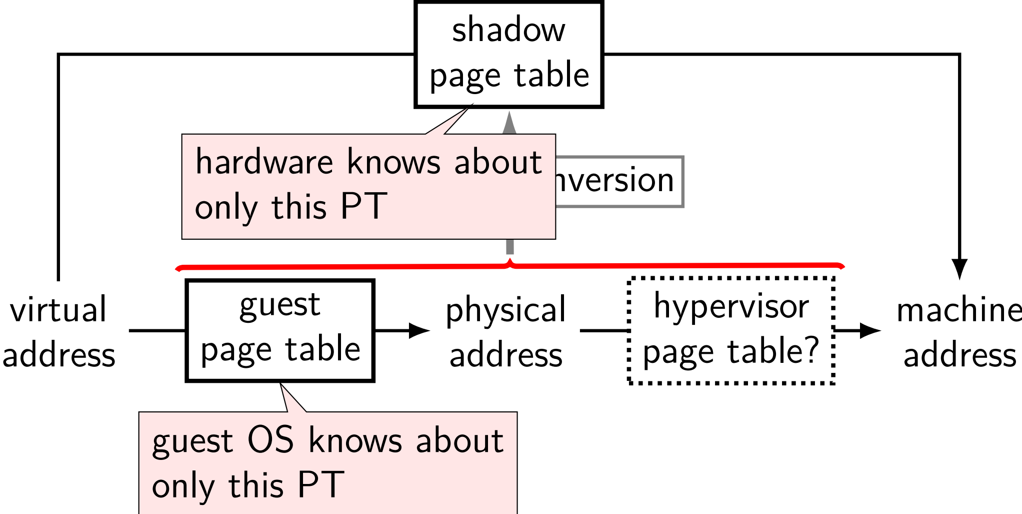
three page tables



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

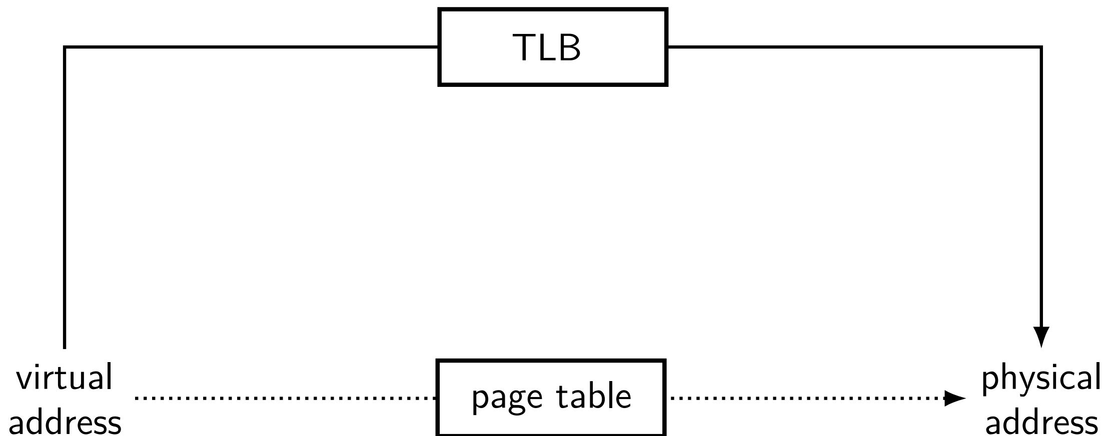
has the same problem

contents synthesized from the 'normal' page table

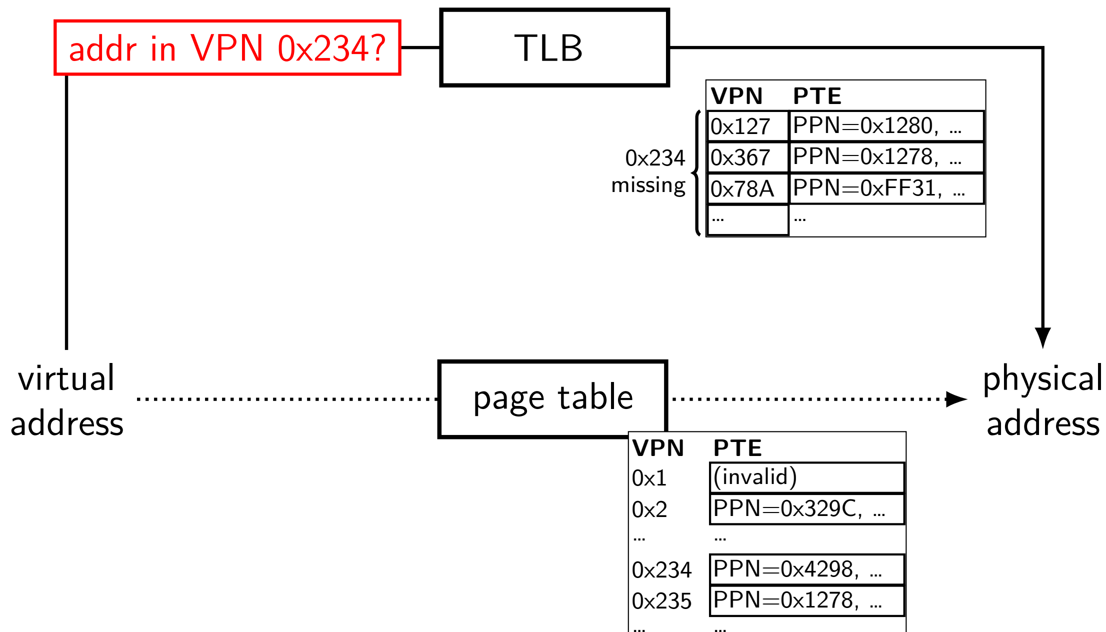
processor needs to decide when to update it

preview: hypervisor can use same solution

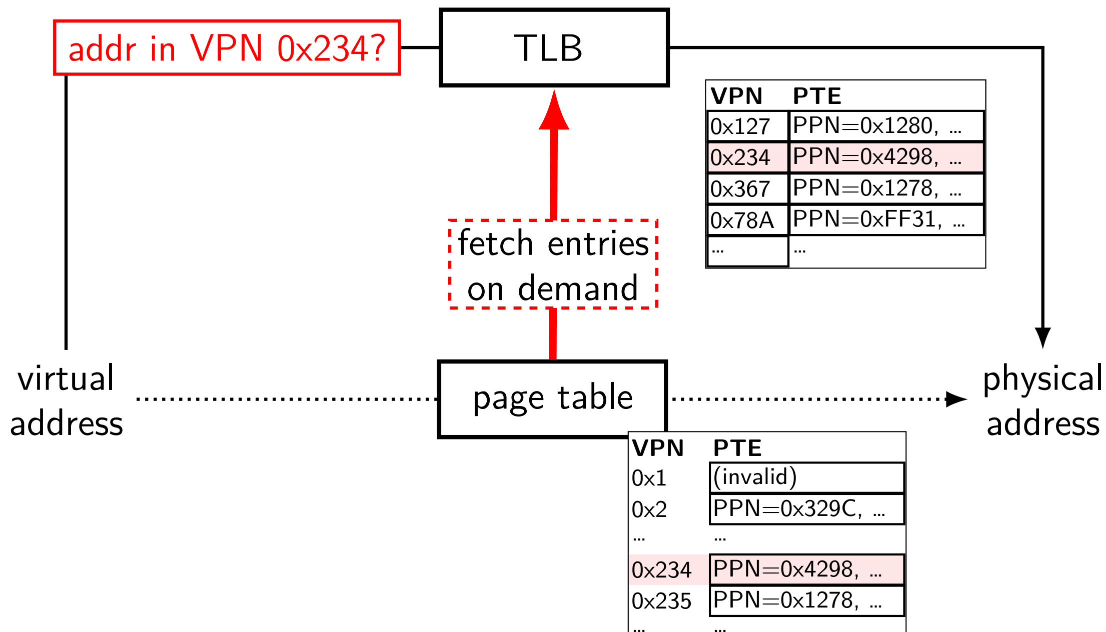
Interlude: TLB (no virtualization)



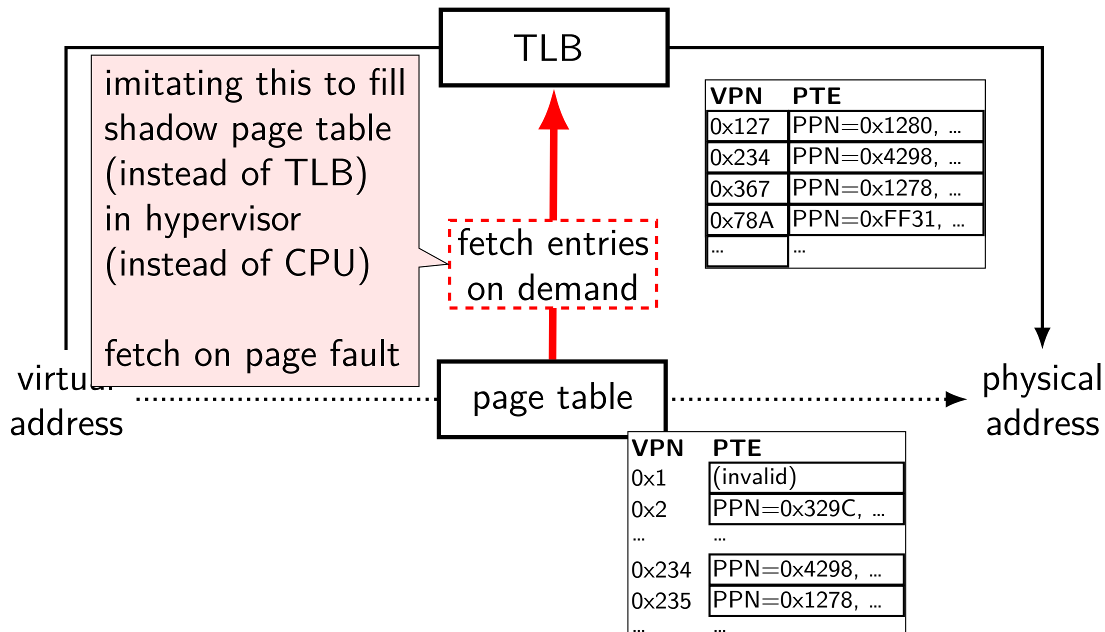
Interlude: TLB (no virtualization)



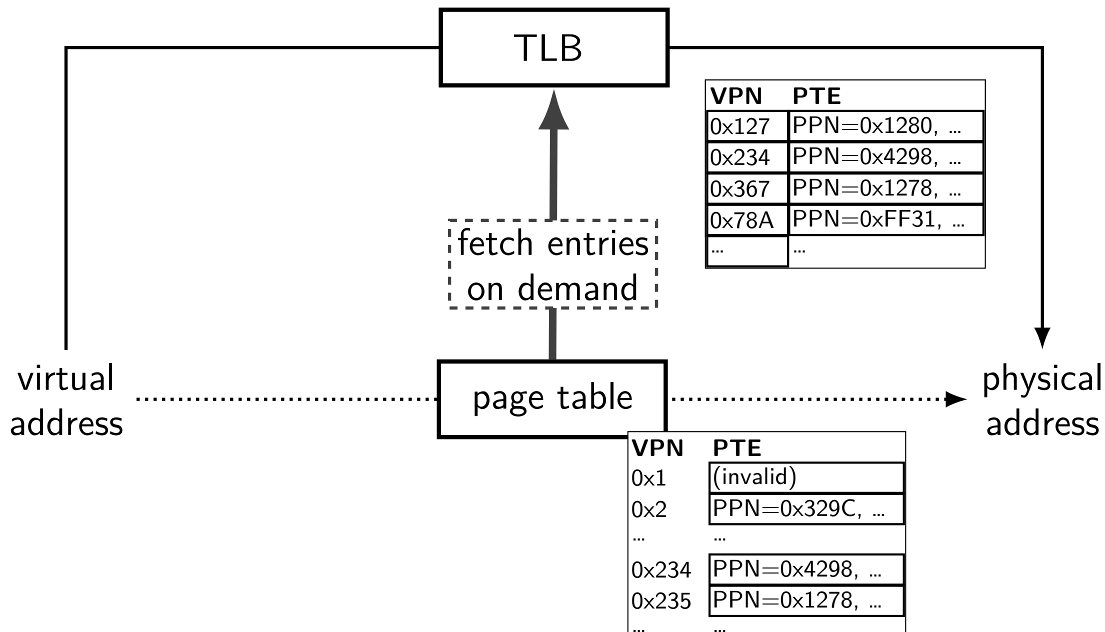
Interlude: TLB (no virtualization)



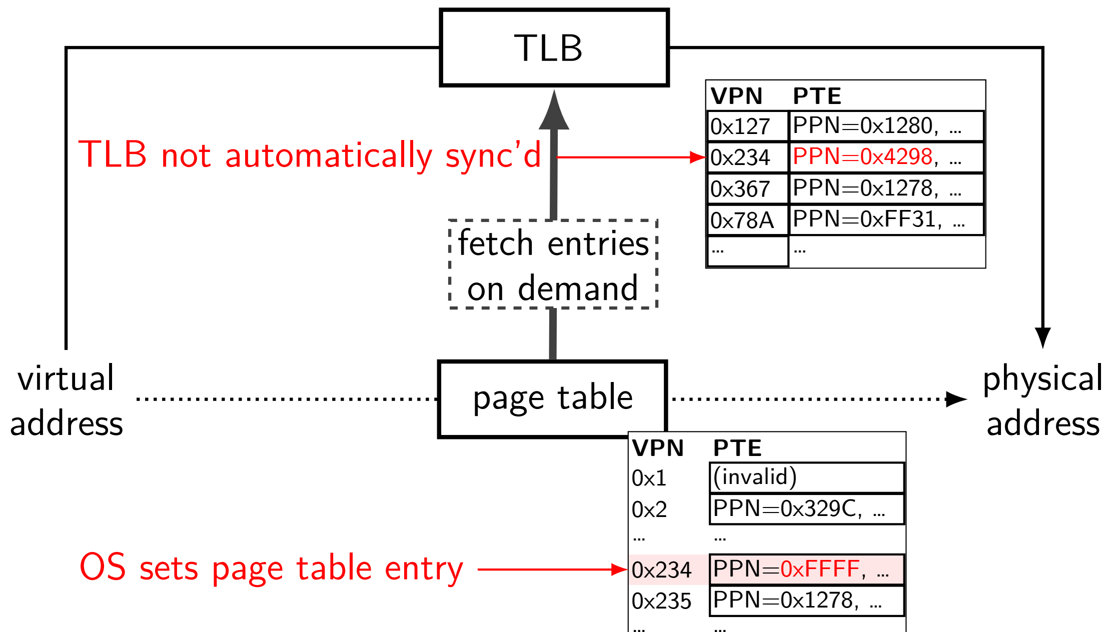
Interlude: TLB (no virtualization)



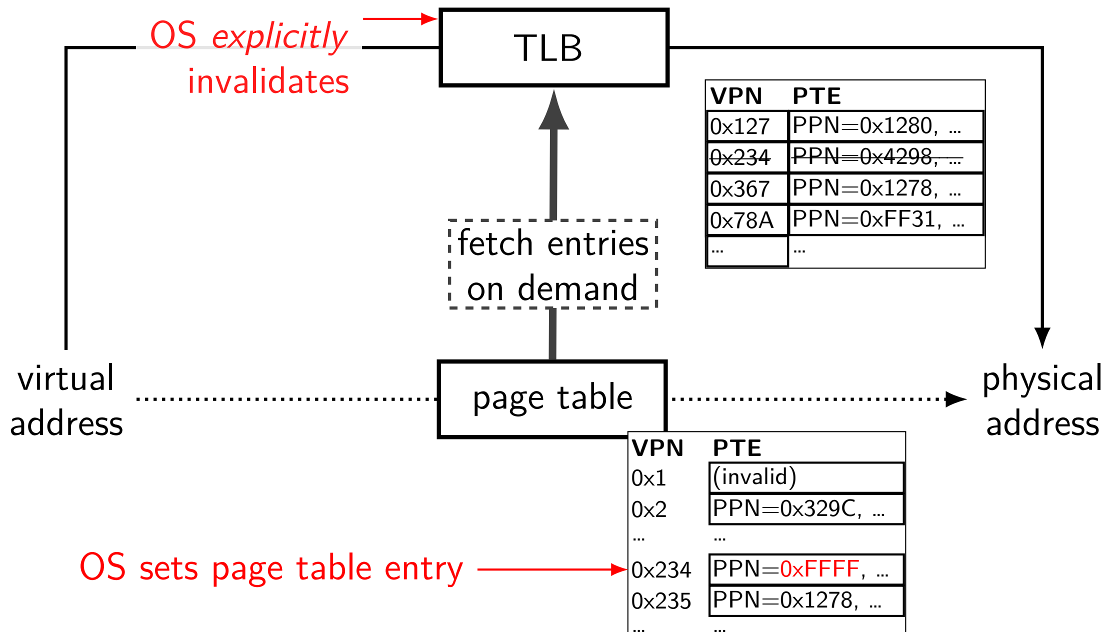
Interlude: TLB (no virtualization)



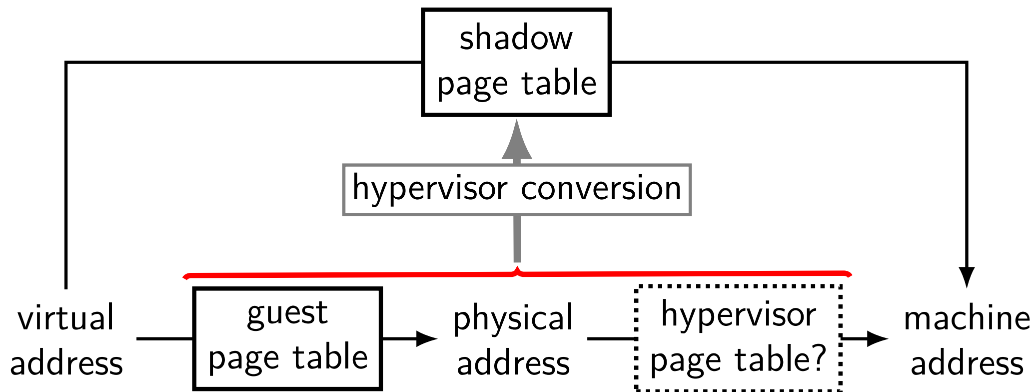
Interlude: TLB (no virtualization)



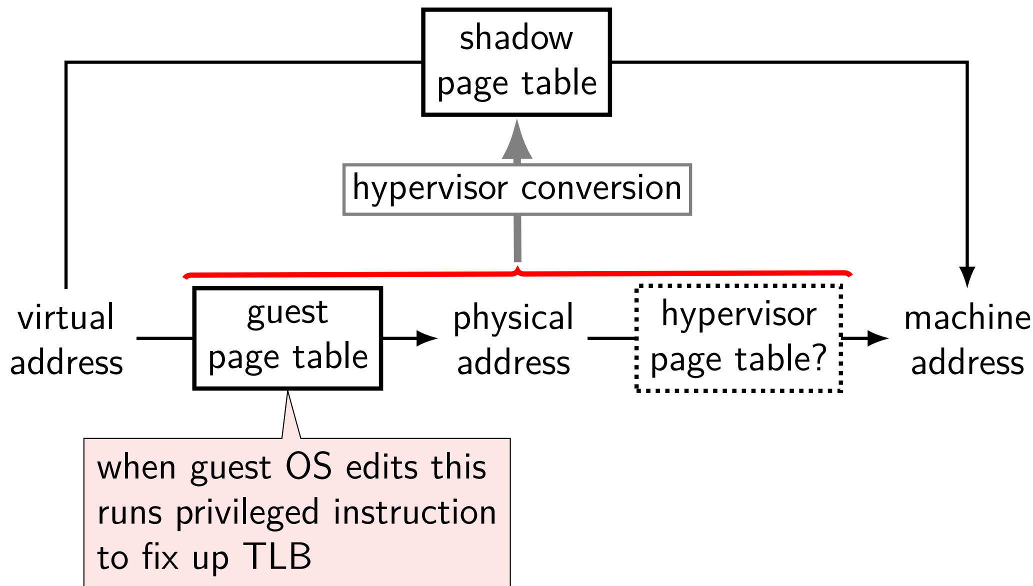
Interlude: TLB (no virtualization)



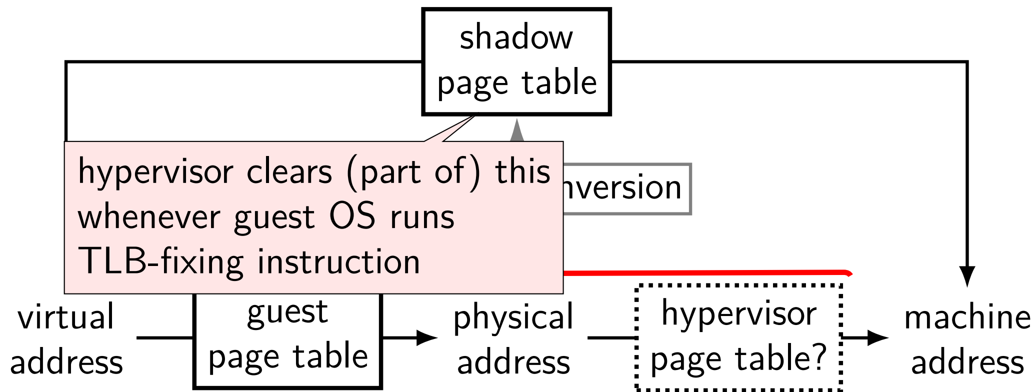
three page tables (revisited)



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

many OSes: invalidate *entire TLB* on context switch

- assumption: TLB only holds entries from one process

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

aside: tagged TLBs

some TLBs support holding entries from multiple page tables

- entries "tagged" with page table they are from

…but not x86 until pretty recently

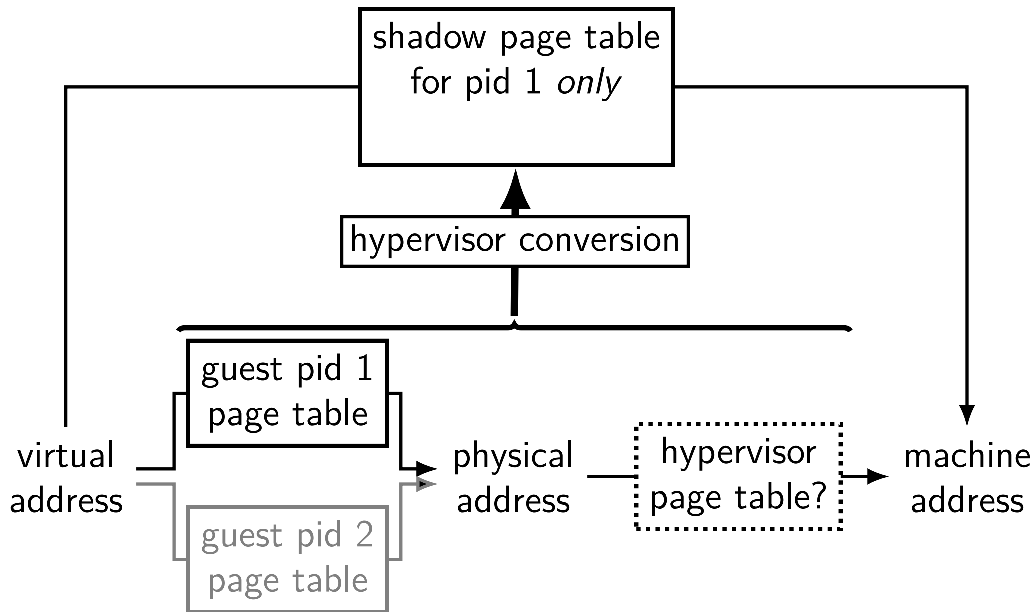
allows OSs to not invalidate entire TLB on context switch

starting to be used by OSes

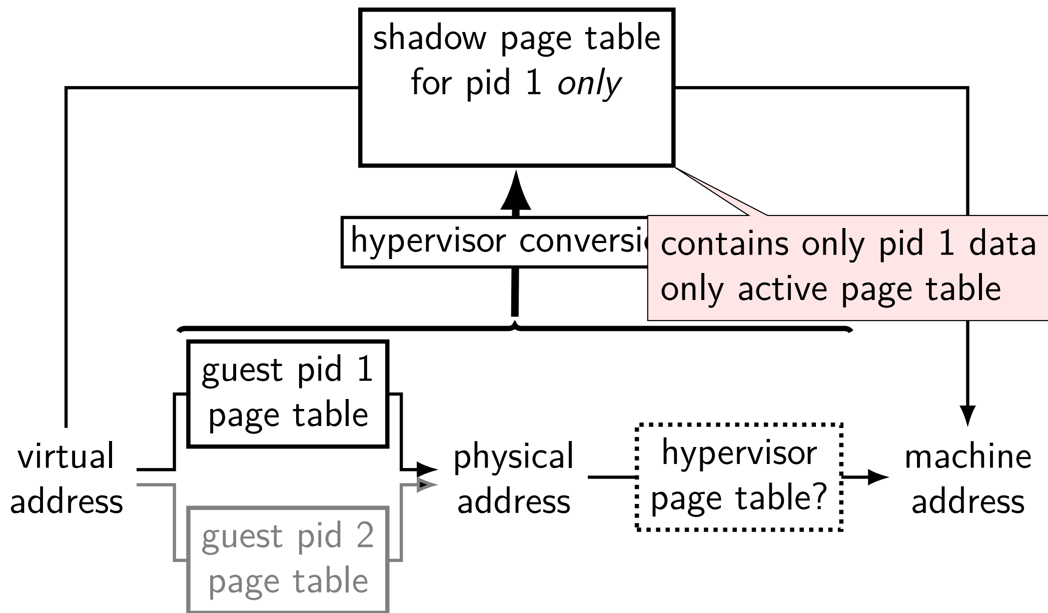
would be really helpful for our virtual machine proposals

- lots of page table switches

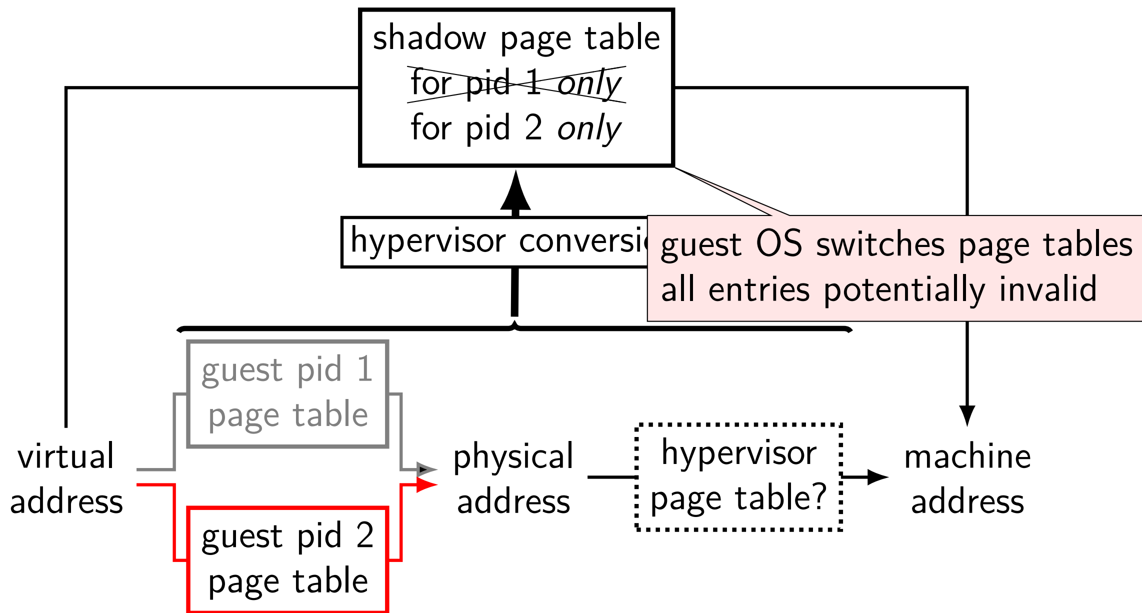
problem with filling on demand



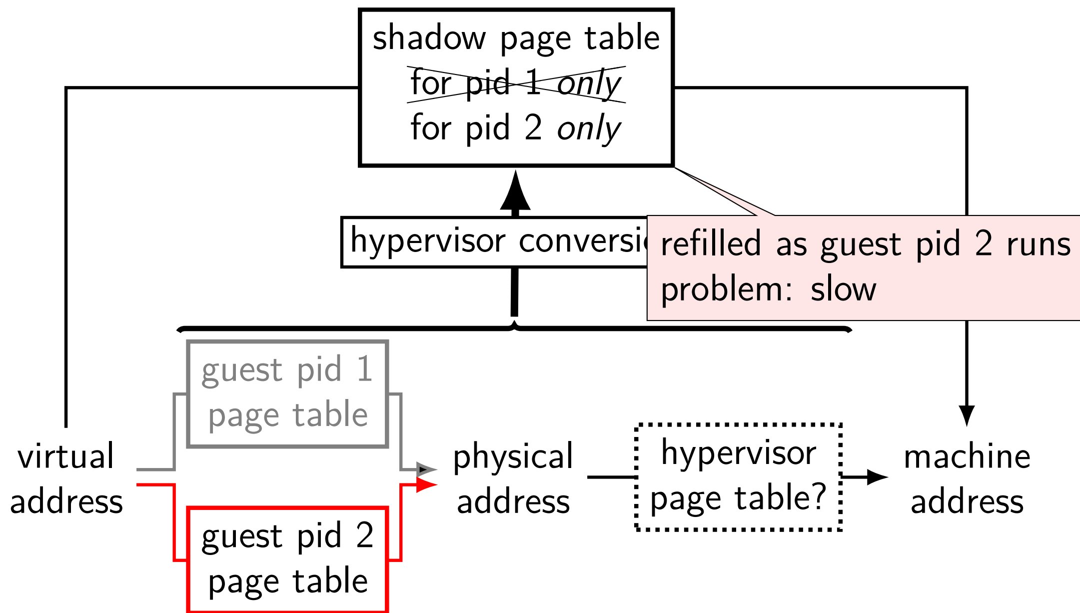
problem with filling on demand



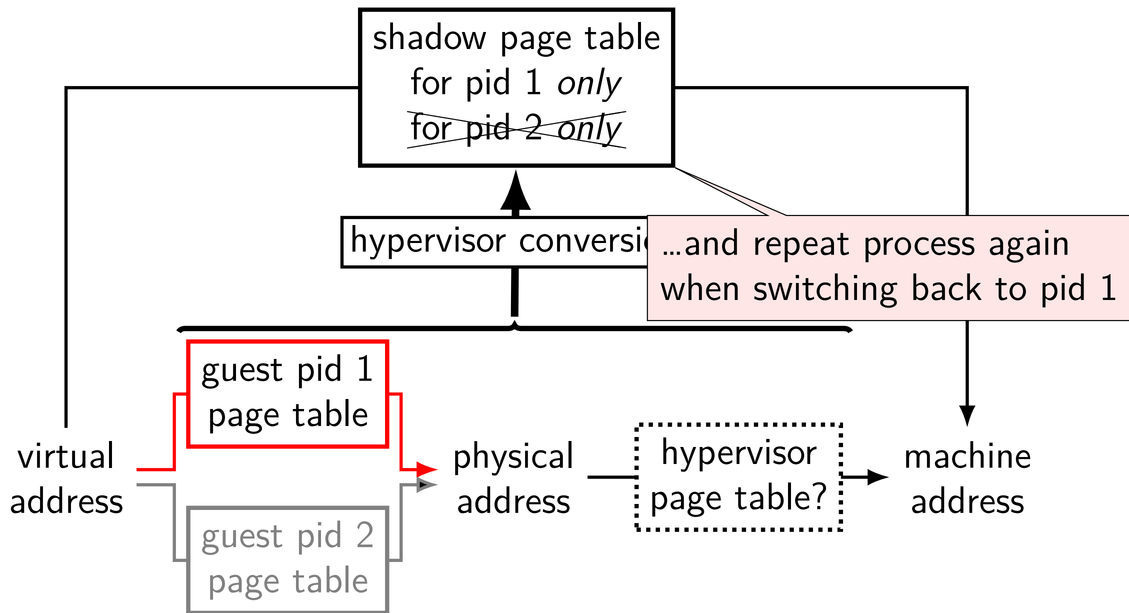
problem with filling on demand



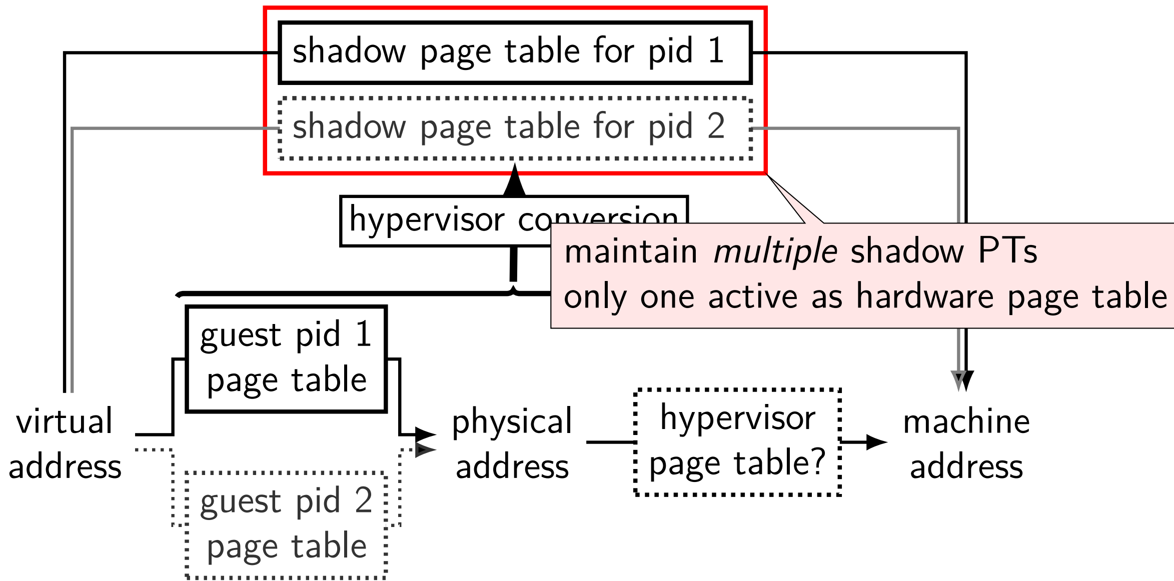
problem with filling on demand



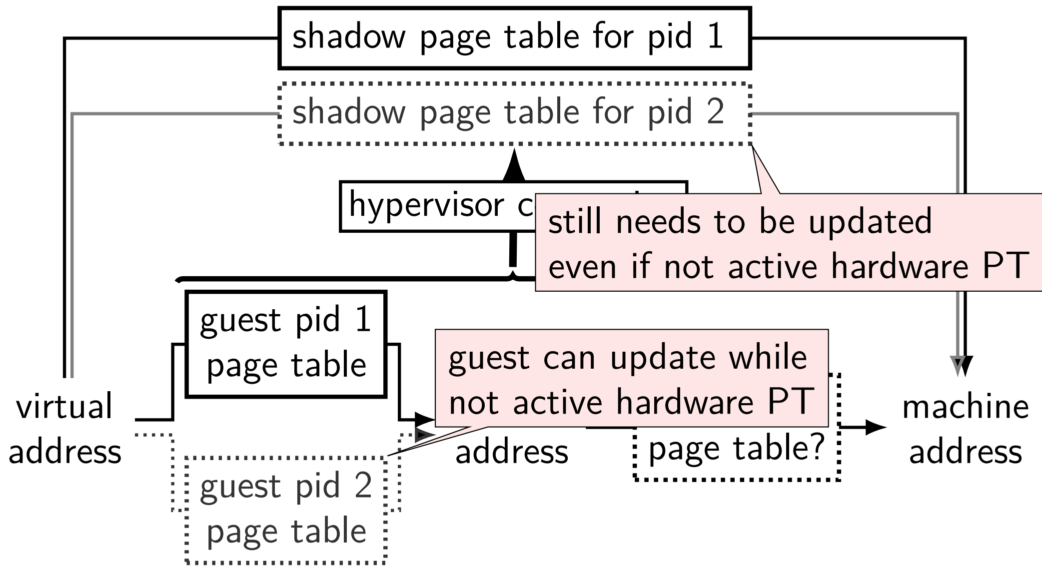
problem with filling on demand



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 guest page tables

(...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

pro: better fit with tagged TLBs

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

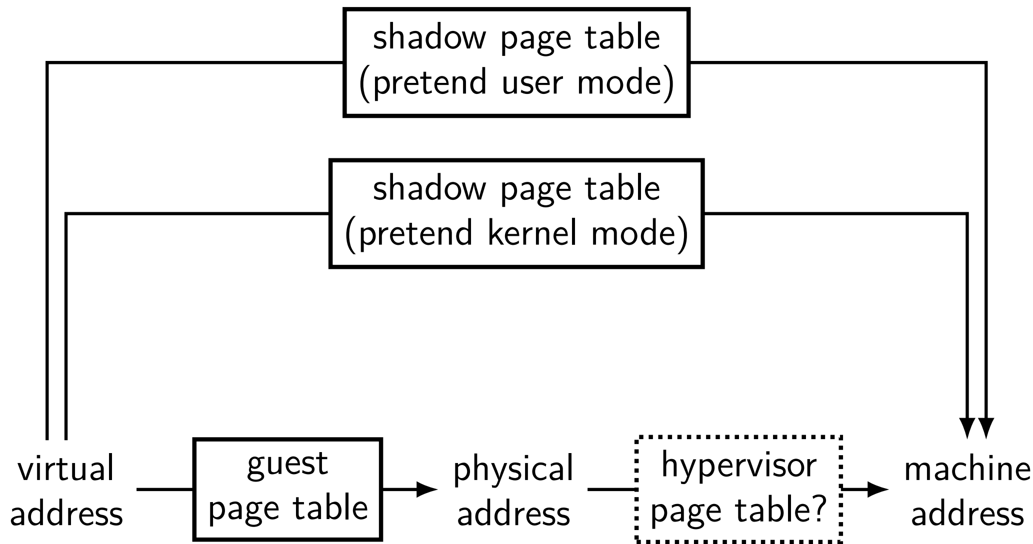
guest OS in pretend kernel mode

- 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?

backup slides

talking to the sandbox

browser kernel sends commands to sandbox

sandbox sends commands to browser kernel

idea: commands only allow necessary things

original Chrome sandbox interface

sandbox to browser “kernel”

- show this image on screen
    - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

browser “kernel” to sandbox

- send user input

original Chrome sandbox interface

sandbox to browser “kernel”

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

browser “

- send user input

needs filtering — at least no `file:` (local file) URLs

original Chrome sandbox interface

sandbox to browser “kernel”

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

browser “kernel” to sa

- send user input

can still read any website!
still sends normal cookies!

original Chrome sandbox interface

sandbox to browser “kernel”

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

> files go to download directory only
> can’t choose arbitrary filenames

browser “kernel” to

- send user input

original Chrome sandbox interface

sandbox to browser "kernel"

- show this image on screen
  - (using shared memory for speed)
- make request for this URL
- download files to local FS
- upload user requested files

browser "kernel" to sandbox

- send user input

browser kernel displays file choser
only permits files selected by user