access control 2 / virtual machines

last time (1)

twophase assignment

protection (mechanism to control access) v security (policy about what's allowed)

access control matrix abstraction

- protection domains (POSIX: user, group IDs) $\times$ objects (files, etc.)
- permitted operations for each pair
- actually storing: access control lists or (domain, object, operations) tuples

access control list

- list of user/group/...+ permitted operations stored with objects (e.g. files)
- POSIX basic permissions: one user + one group + everyone else
  - read/write/execute specified for each; chmod
- POSIX full ACLs: list of arbitrarily many users, groups

last time (2)

enforcing permissions: check in system calls

superuser (POSIX user ID 0)

- permissions check special case: always allow ID 0
- can setuid to change current user ID — how 'login' works

set-user-ID programs

sudo

```
tj1a@somemachine$ sudo restart
Password: *********
```

sudo: run command with superuser permissions

started by non-superuser

recall: inherits non-superuser UID

can't just call `setuid(0)`

set-user-ID sudo

extra metadata bit on *executables*: set-user-ID

if set: `exec()` syscall changes effective user ID to owner's ID

`sudo` program: owned by root, marked set-user-ID

marking setuid: `chmod u+s`

set-user ID gates

set-user ID program: gate to higher privilege

controlled access to extra functionality

make authorization/authentication decisions *outside the kernel*

way to allow normal users to do *one thing that needs privileges*

- write program that does that one thing — nothing else!
- make it owned by user that can do it (e.g. root)
- mark it set-user-ID

want to allow only some user to do the thing

- make program check which user ran it

uses for setuid programs

mount USB stick

- setuid program controls option to kernel mount syscall
- make sure user can't replace sensitive directories
- make sure user can't mess up filesystems on normal hard disks
- make sure user can't mount new setuid root files

control access to device — printer, monitor, etc.

- setuid program talks to device + decides who can

write to secure log file

- setuid program ensures that log is append-only for normal users

bind to a particular port number < 1024

- setuid program creates socket, then becomes not root

set-user-ID program v syscalls

hardware decision: some things only for kernel

system calls: *controlled* access to things kernel can do

decision about how can do it: in the kernel

kernel decision: some things only for root (or other user)

set-user-ID programs: controlled access to things root/… can do

decision about how can do it: made by root/…

a broken setuid program: setup

suppose I have a directory all-grades on shared server

in it I have a folder for each assignment

and within that a text file for each user's grade + other info

say I don't have flexible ACLs and want to give each user access

a broken setuid program: setup

suppose I have a directory all-grades on shared server

in it I have a folder for each assignment

and within that a text file for each user's grade + other info

say I don't have flexible ACLs and want to give each user access

one (bad?) idea: setuid program to read grade for assignment

`./print_grade assignment`

outputs grade from `all-grades/assignment/USER.txt`

a very broken setuid program

print_grade.c:

```
int main(int argc, char **argv) {
    char filename[500];
    sprintf(filename, "all-grades/%s/%s.txt",
            argv[1], getenv("USER"));
    int fd = open(filename, O_RDWR);
    char buffer[1024];
    read(fd, buffer, 1024);
    printf("%s: %s\n", argv[1], buffer);
}
```

HUGE amount of stuff can go wrong

examples?

set-user ID programs are very hard to write

what if stdin, stdout, stderr start closed?

what if the PATH env. var. set to directory of malicious programs?

what if `argc == 0`?

what if dynamic linker env. vars are set?

what if some bug allows memory corruption?

…

a delegation problem

consider printing program marked setuid to access printer

- decision: no accessing printer directly
- printing program enforces page limits, etc.

command line: file to print

can printing program just call open()?

a broken solution

```
if (original user can read file from argument) {
    open(file from argument);
    read contents of file;
    write contents of file to printer
    close(file from argument);
}
```

hope: this prevents users from printing files than can't read

problem: race condition!

a broken solution / why

setuid program	other user program
	create normal file `toprint.txt`
check: can user access? (yes)	—
	`unlink("toprint.txt")`
	`link("/secret", "toprint.txt")`
`open("toprint.txt")`	—
read ...	—

link: create new directory entry for file

- another option: rename, symlink ("symbolic link" — alias for file/directory)
- another possibility: run a program that creates secret file (e.g. temporary file used by password-changing program)

time-to-check-to-time-of-use vulnerability

TOCTTOU solution

temporarily ‘become’ original user

then open

then turn back into set-uid user

this is why POSIX processes have multiple user IDs

can swap out effective user ID temporarily

practical TOCTTOU races?

can use symlinks *maze* to make check slower

symlink `toprint.txt` → `a/b/c/d/e/f/g/normal.txt`
symlink `a/b` → `../a`
symlink `a/c` → `../a`
...

lots of time spent following symbolic links when program opening toprint.txt

gives more time to sneak in unlink/link or (more likely) rename

exercise

which (if any) of the following would fix for a TOCTTOU vulnerability in our setuid printing application? (assume the Unix-permissions without ACLs are in use)

[A] **both before and after** opening the path passed in for reading, check that the path is accessible to the user who ran our application

[B] after opening the path passed in for reading, using `fstat` with the file descriptor opened to check the permissions on the file

[C] before opening the path, verify that the user controls the file referred to by the path **and** the directory containing it

some security tasks (1)

helping students collaborate in ad-hoc small groups on shared server?

Q1: what to allow/prevent?

Q2: how to use POSIX mechanisms to do this?

some security tasks (2)

letting students assignment files to faculty on shared server?

Q1: what to allow/prevent?

Q2: how to use POSIX mechanisms to do this?

some security tasks (3)

running untrusted game program from Internet?

Q1: what to allow/prevent?

Q2: how to use POSIX mechanisms to do this?

ambient authority

POSIX permissions based on user/group IDs process has

- correct user/group ID — can read file
- correct user ID — can kill process

permission information "on the side"

- separate from how to identify file/process

sometimes called *ambient authority*

"there's authorization in the air..."

alternate approach: ability to address = permission to access

capabilities

token to identify = permission to access

(typically *opaque* token)

capabilities

token to identify = permission to access

(typically *opaque* token)

pro: "what object is this token" check = "can access" check:

simpler?

some capability list examples

file descriptors

- list of open files process has access 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 access 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

some ways of sharing capabilities:

inherited by spawned programs

- file descriptors/page tables do this

send over local socket or pipe

- Unix: usually supported for file descriptors!
- (look up `SCM_RIGHTS` — slightly 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()`

recall: the virtual machine interface

application

operating system — virtual machine interface

hardware — physical machine interface

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)
- I'll often talk as if hypervisor is OS to keep things simpler
  - if hypervisor not OS: host OS will provide new system calls/etc.

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
- application should still be unmodified

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

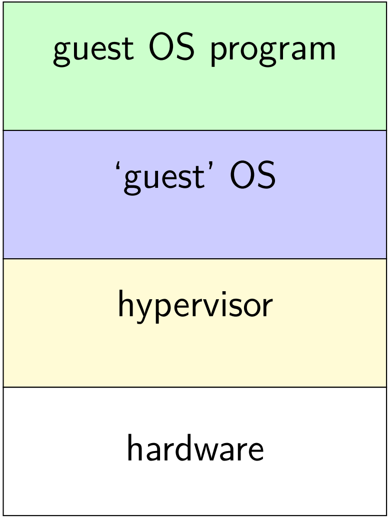
one variation: extra HW support for VMs (if time)

one variation: compile guest OS machine code to new machine code

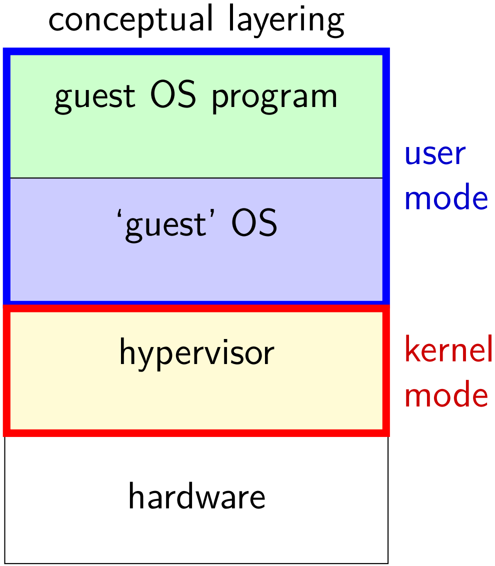
- not as slow as you'd think, sometimes

VM layering (intro)

conceptual layering



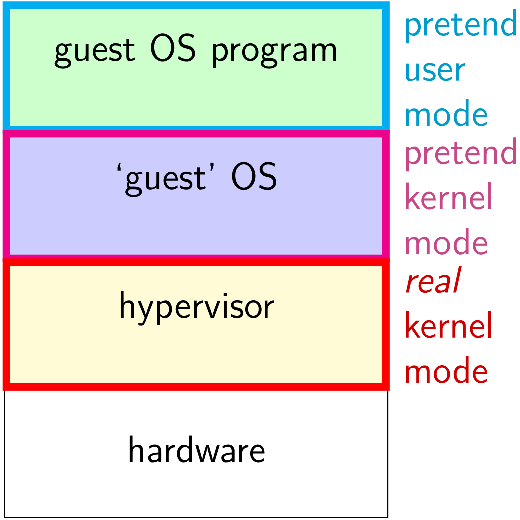
VM layering (intro)



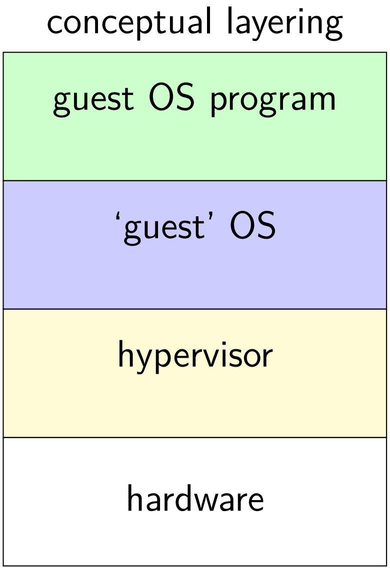
≈ hypervisor's process

VM layering (intro)

conceptual layering

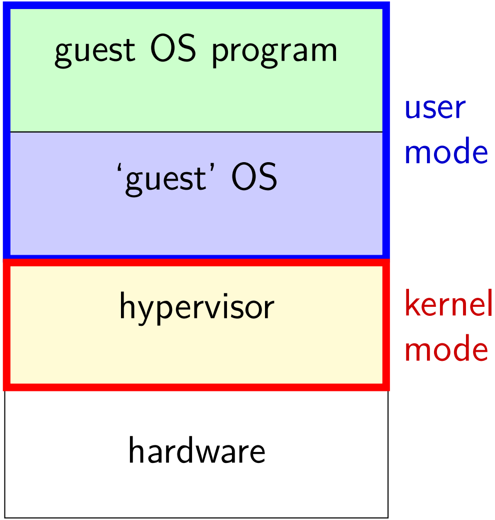


VM layering


conceptual layering
guest OS program
'guest' OS
hypervisor
hardware

VM layering

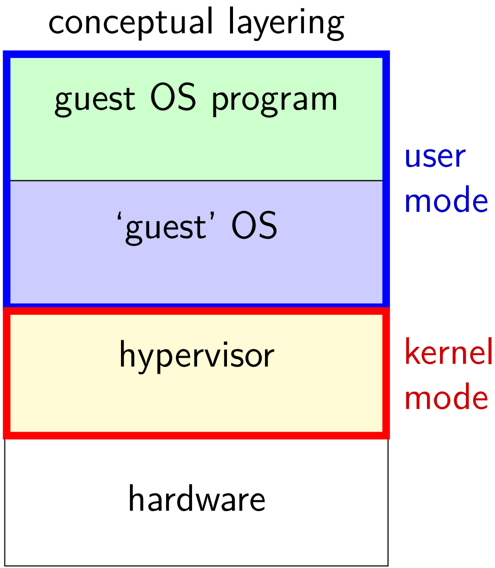
conceptual layering



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...

VM layering

conceptual layering

guest OS program — pretend user mode

'guest' OS — pretend kernel mode

hypervisor — *real* kernel mode

hardware

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
guest OS exception table ptr
...

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

VM layering

conceptual layering

guest OS program	pretend user mode
'guest' OS	pretend kernel mode
hypervisor	*real* kernel mode
hardware	

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
guest OS exception table ptr
... virtual machine state

real ("shadow") 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/special 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/special instructions trigger fault

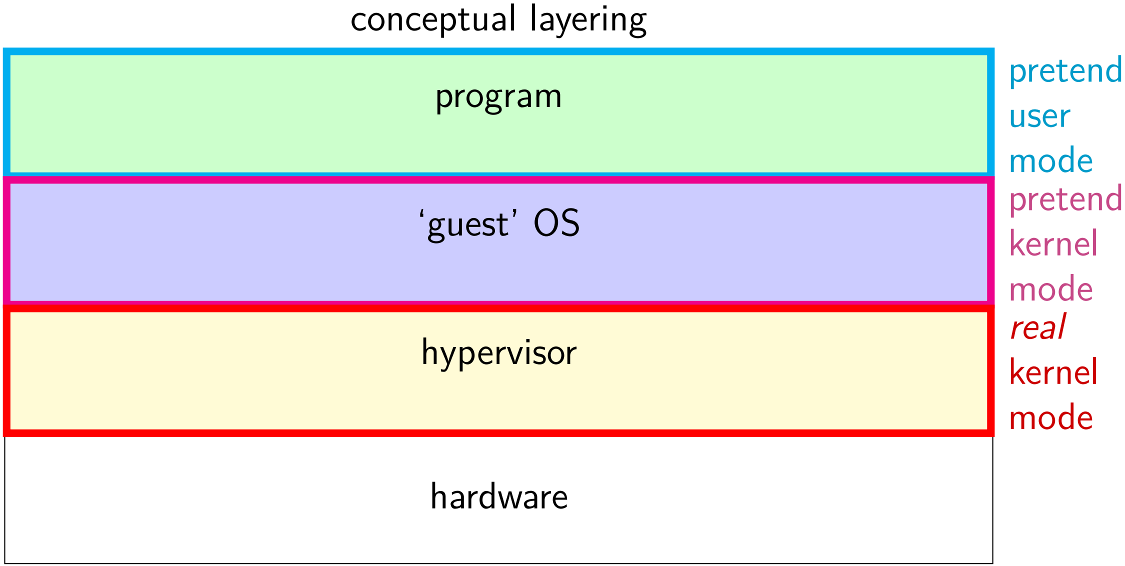
- e.g. accessing device memory directly (page fault)
- e.g. changing the exception table (protection fault)

normal OS: crash the program

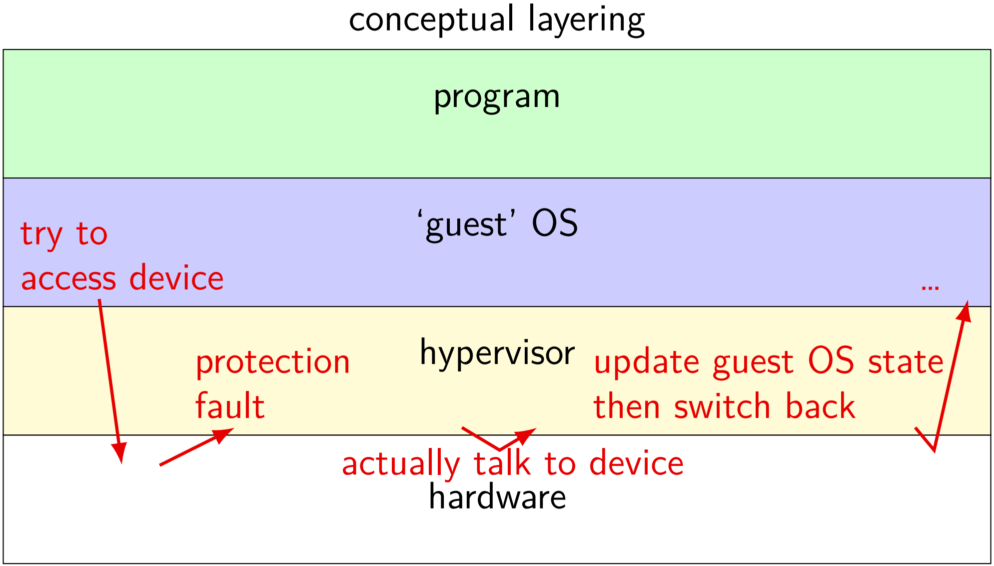
hypervisor: pretend it did the right thing

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

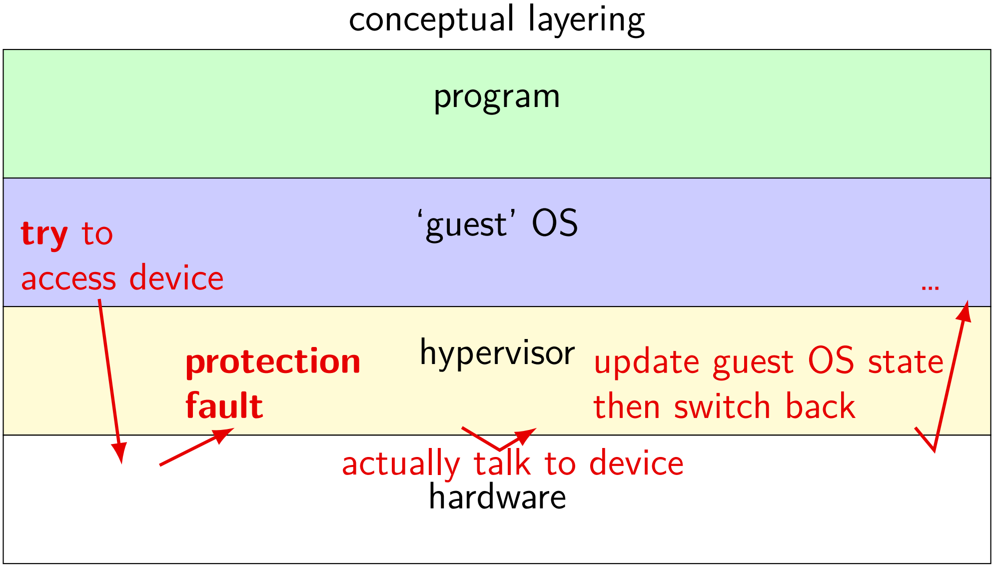
privileged I/O flow



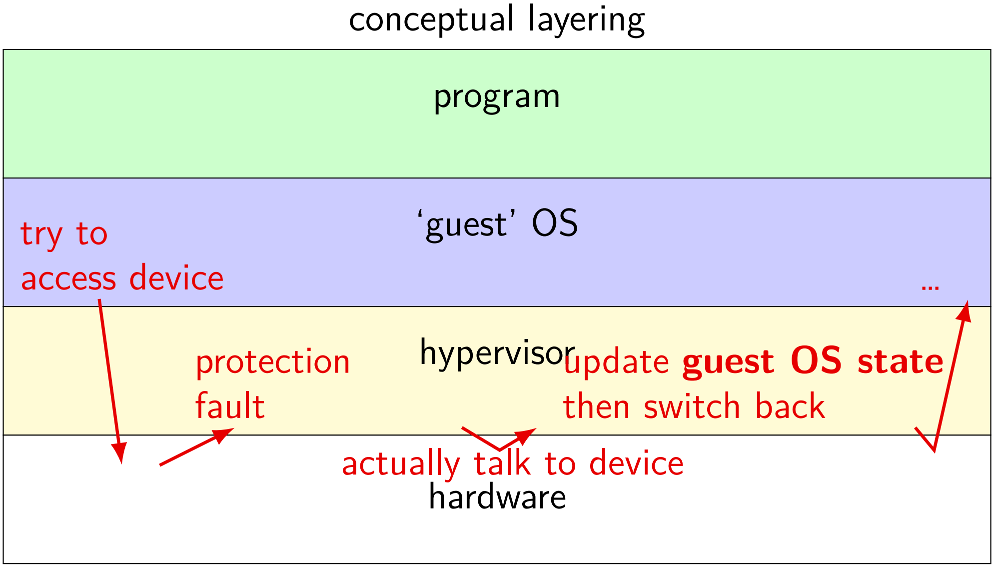
privileged I/O flow



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/special 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 (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: system calls

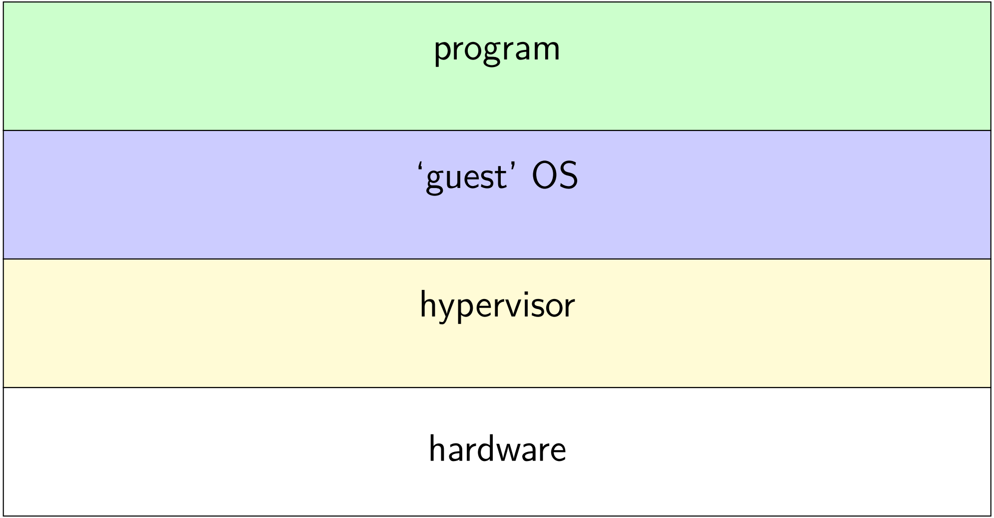
system calls special case of privileged instruction:

system call exception:

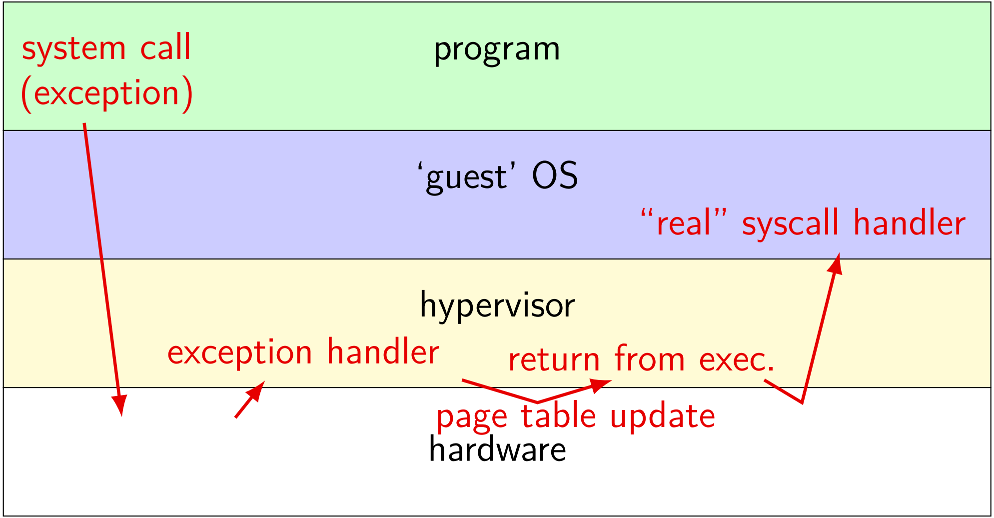
- pretend user mode: execute guest OS's system call handler
- pretend kernel mode: execute guest OS's system call handler

returning from system call? priviliged operation to emulate

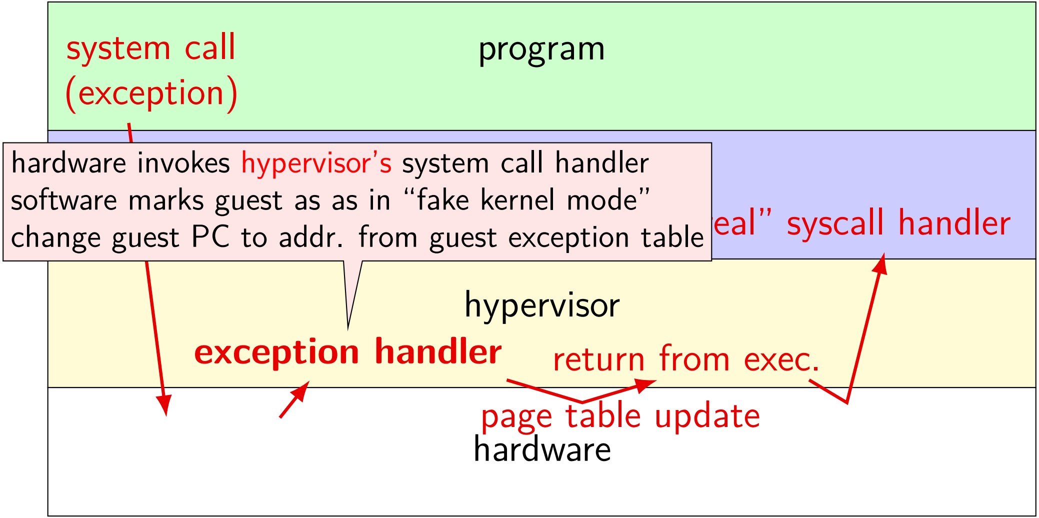
system call/exception flow (part 1)



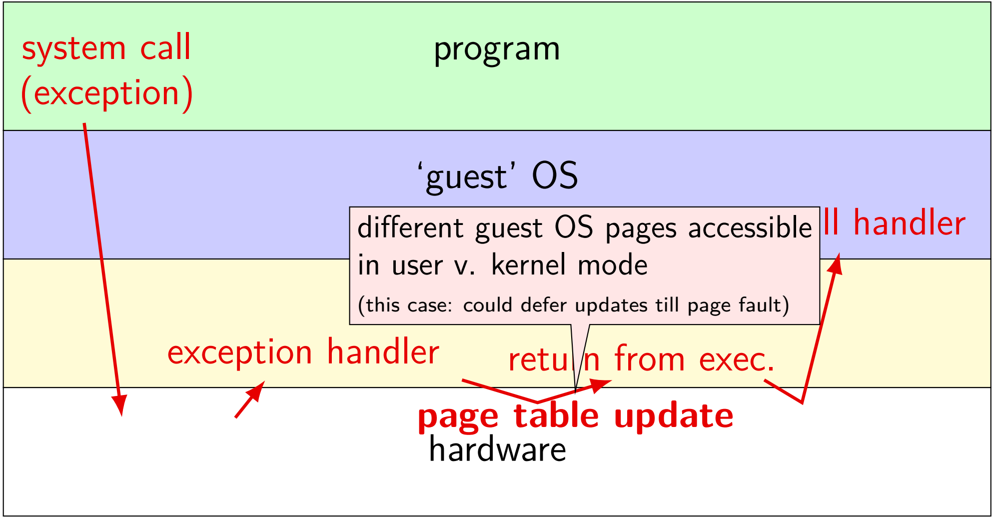
system call/exception flow (part 1)



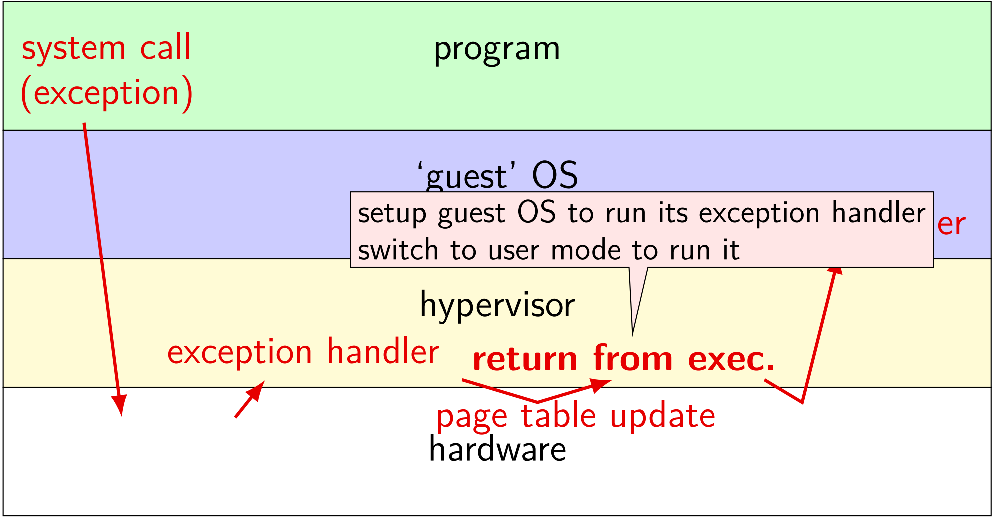
system call/exception flow (part 1)



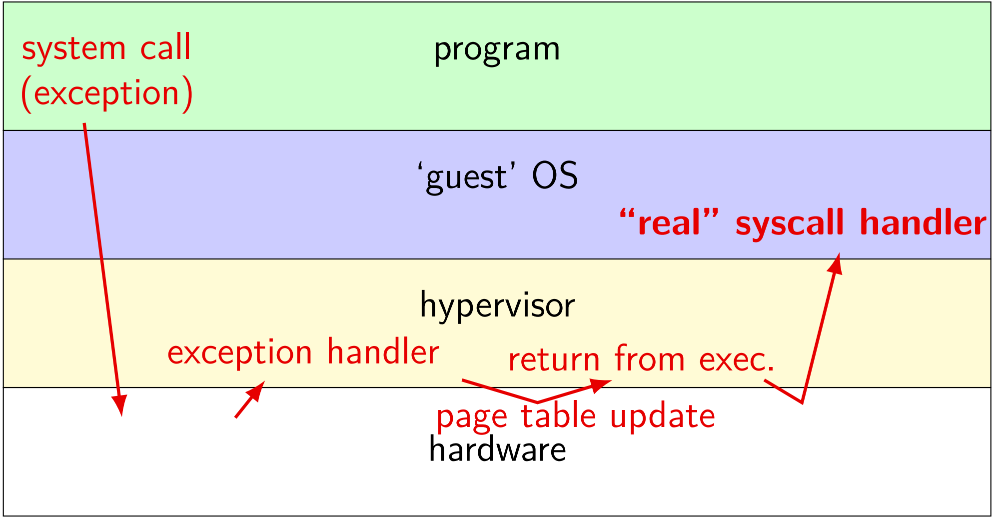
system call/exception flow (part 1)



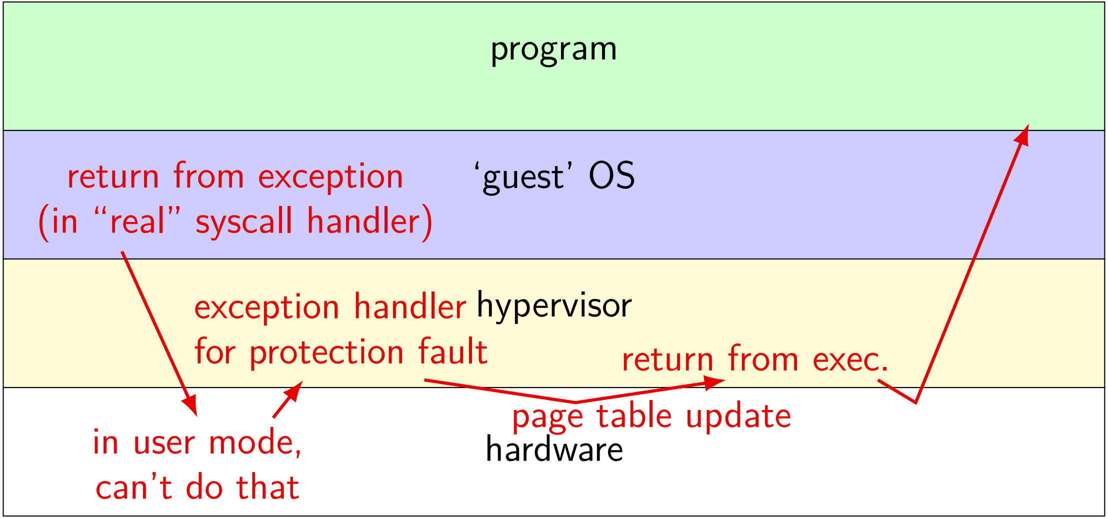
system call/exception flow (part 1)



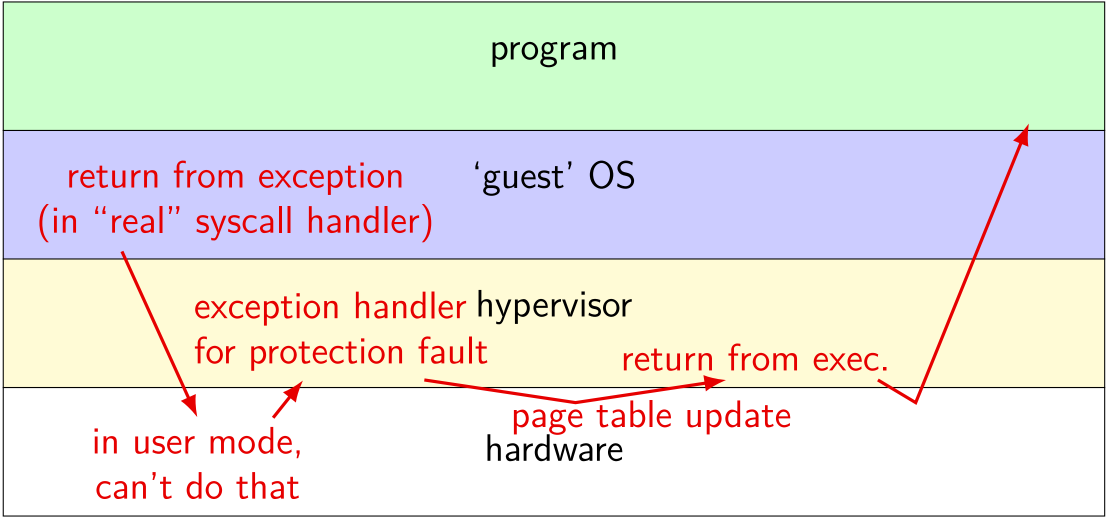
system call/exception flow (part 1)



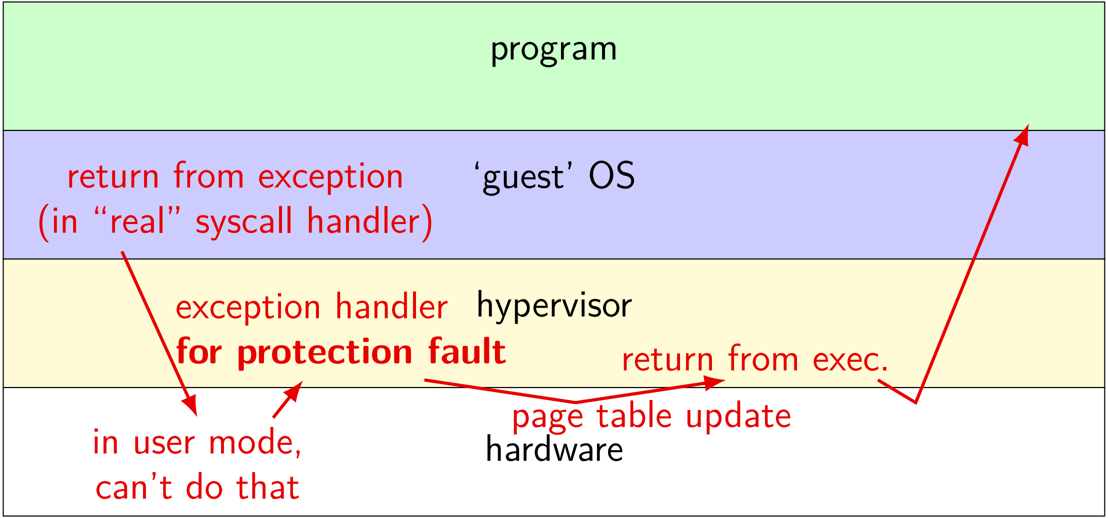
system call/exception flow (part 2)



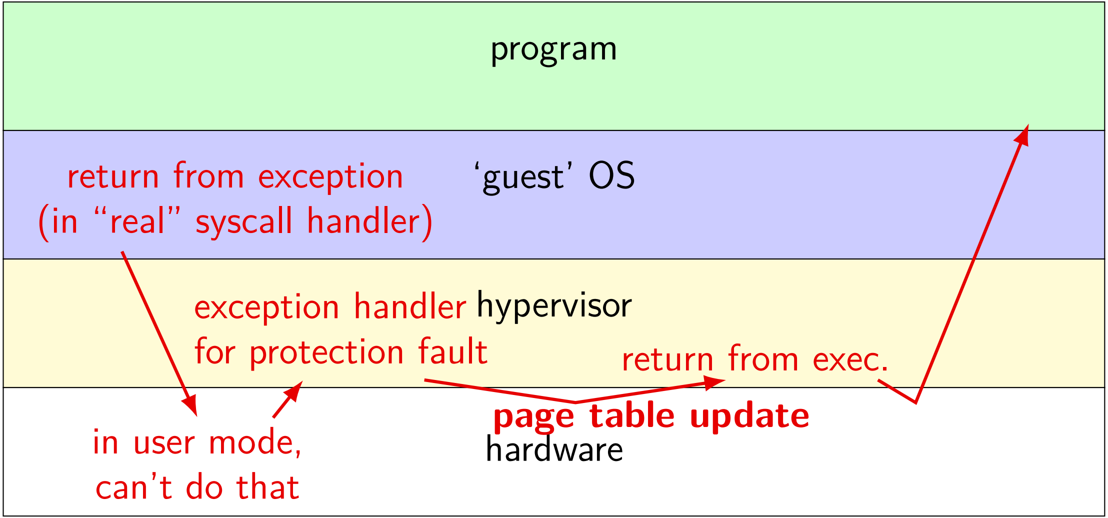
system call/exception flow (part 2)



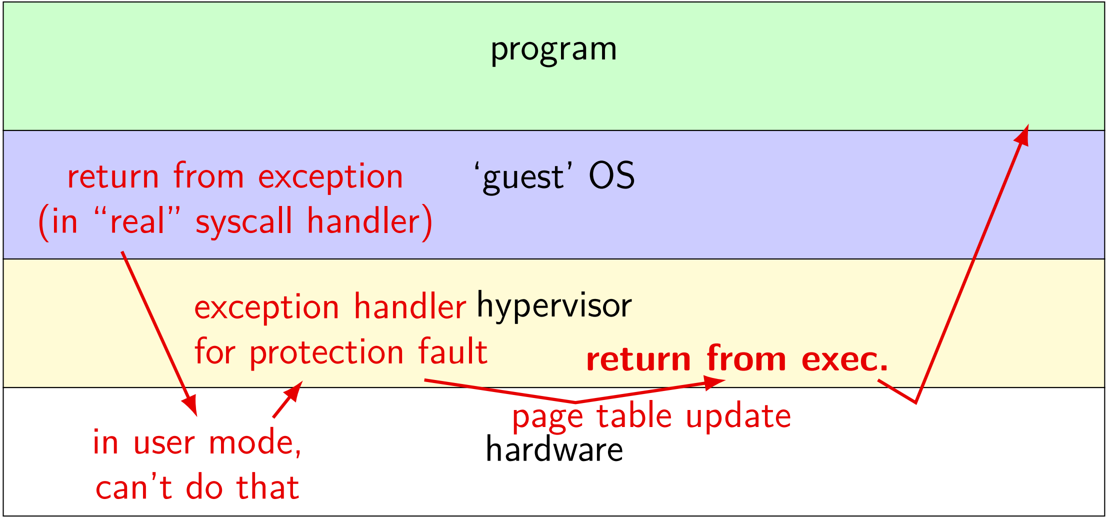
system call/exception flow (part 2)



system call/exception flow (part 2)



system call/exception flow (part 2)



trap and emulate (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)

exercise

guest OS running user program

makes system call write system call to write 4 characters to screen

write system call implementation does write by writing character at a time to memory mapped I/O address

how many exceptions occur on the real hardware?

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

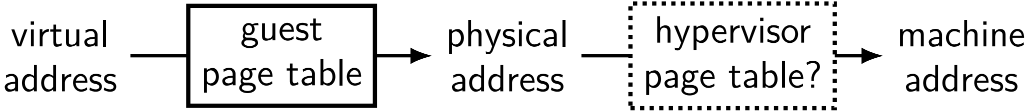
terms for this lecture

virtual address — virtual address for guest OS

physical address — physical address for guest OS

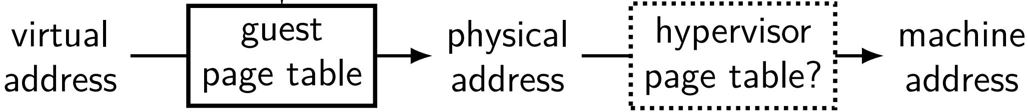
machine address — physical address for hypervisor/host OS

three page tables



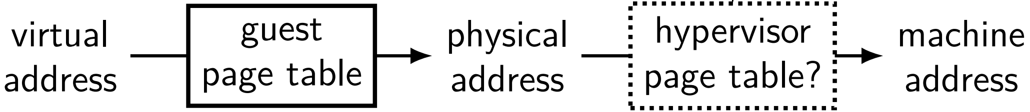
three page tables

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

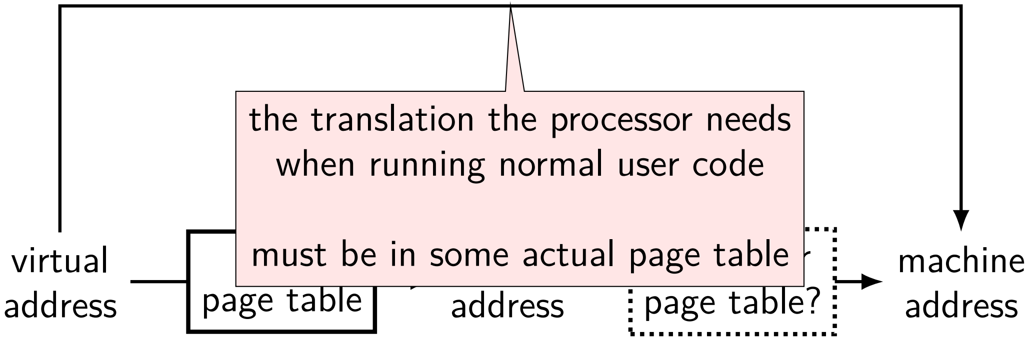


three page tables

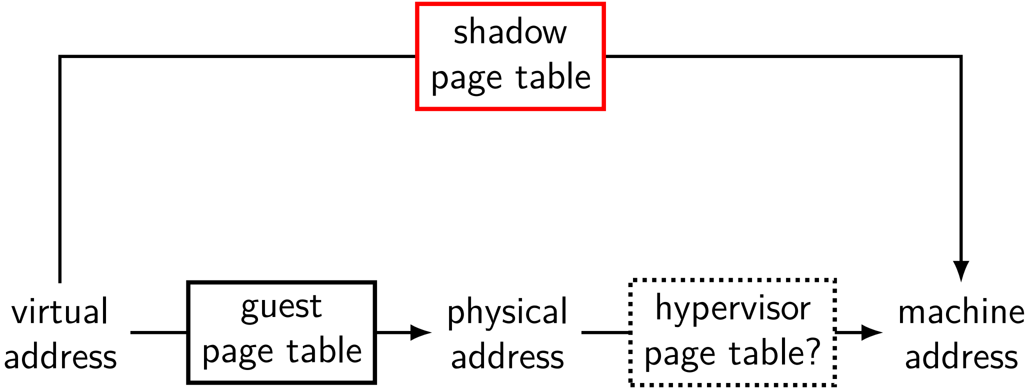
need to allow OS to use any address
run multiple guests in same memory
dynamically allocate memory
normally: use page table for this



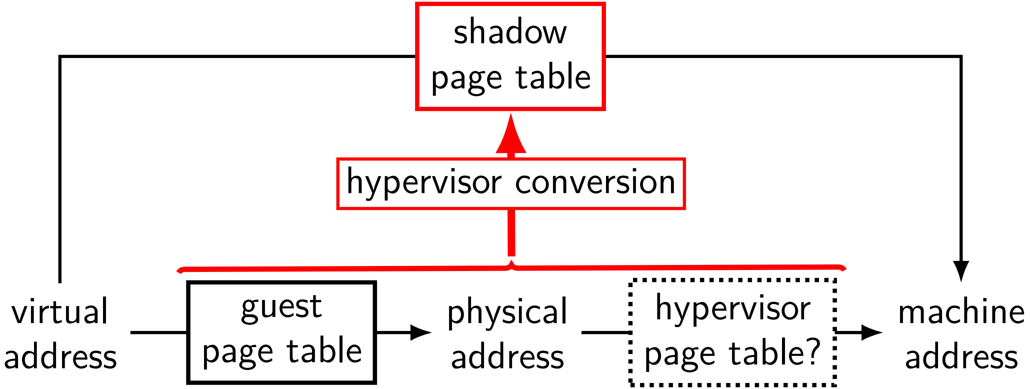
three page tables



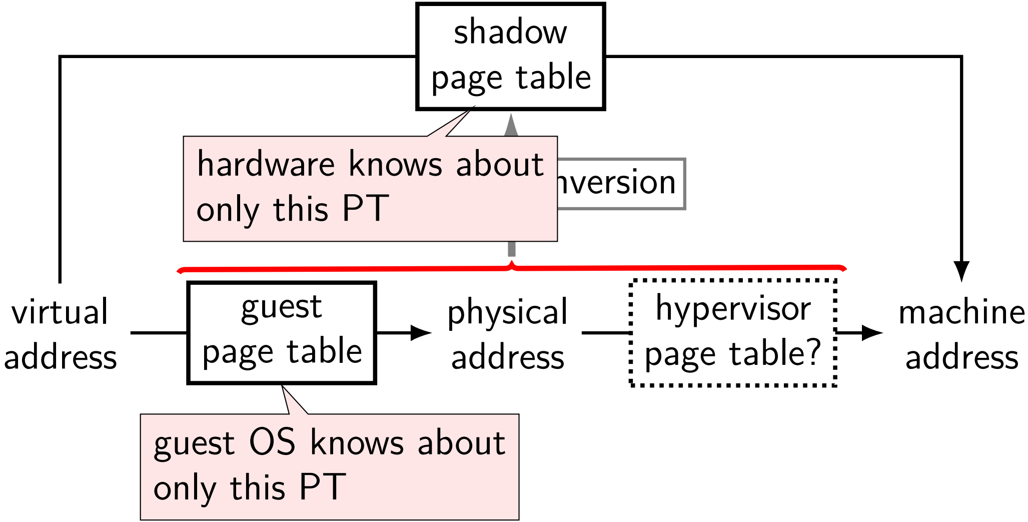
three page tables



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

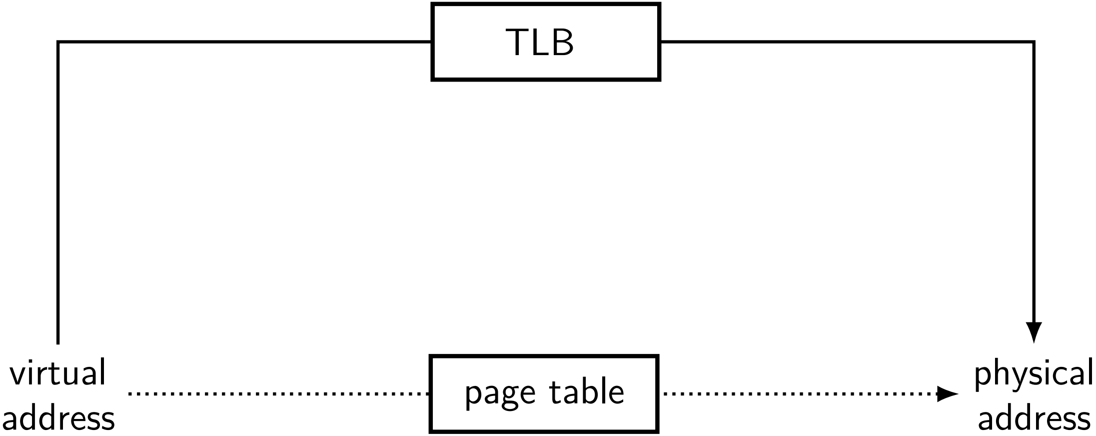
has the same problem

contents synthesized from the 'normal' page table

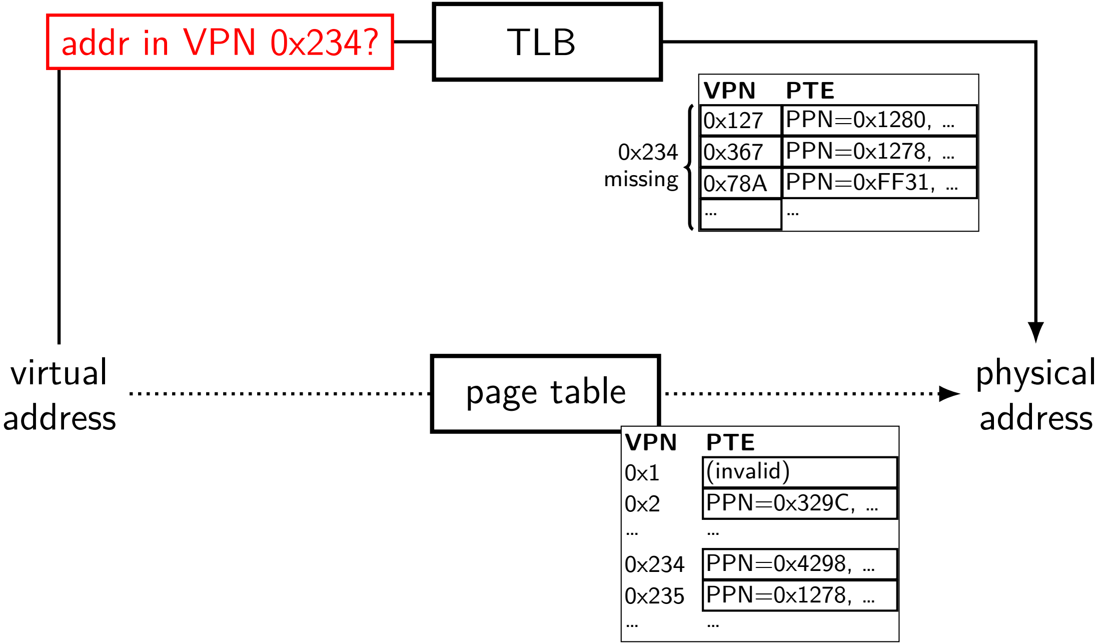
processor needs to decide when to update it

preview: hypervisor can use same solution

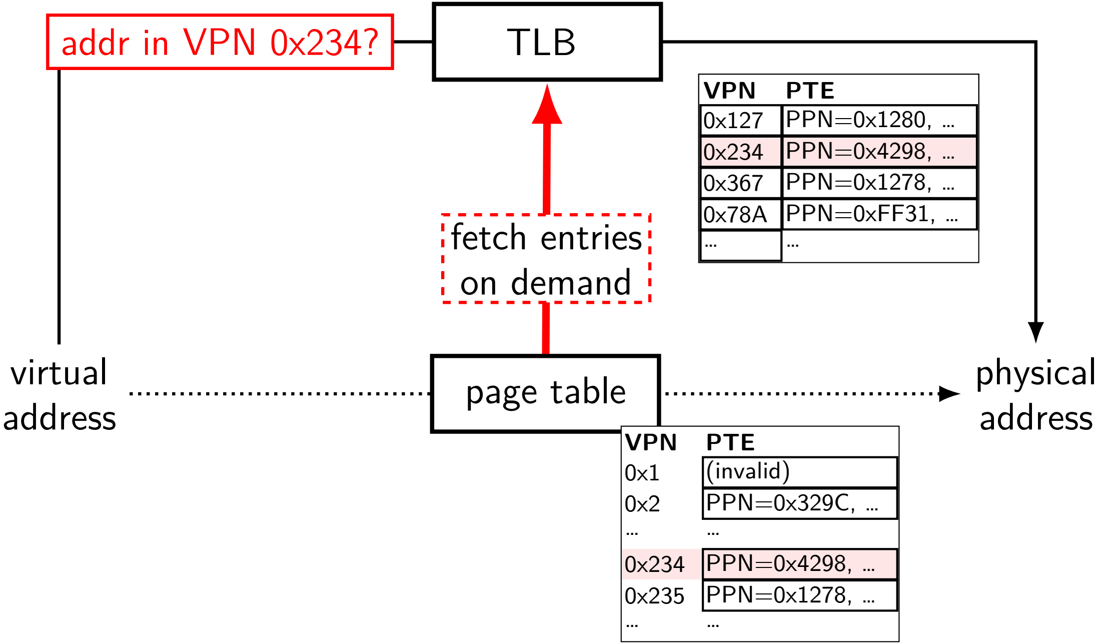
Interlude: TLB (no virtualization)



Interlude: TLB (no virtualization)



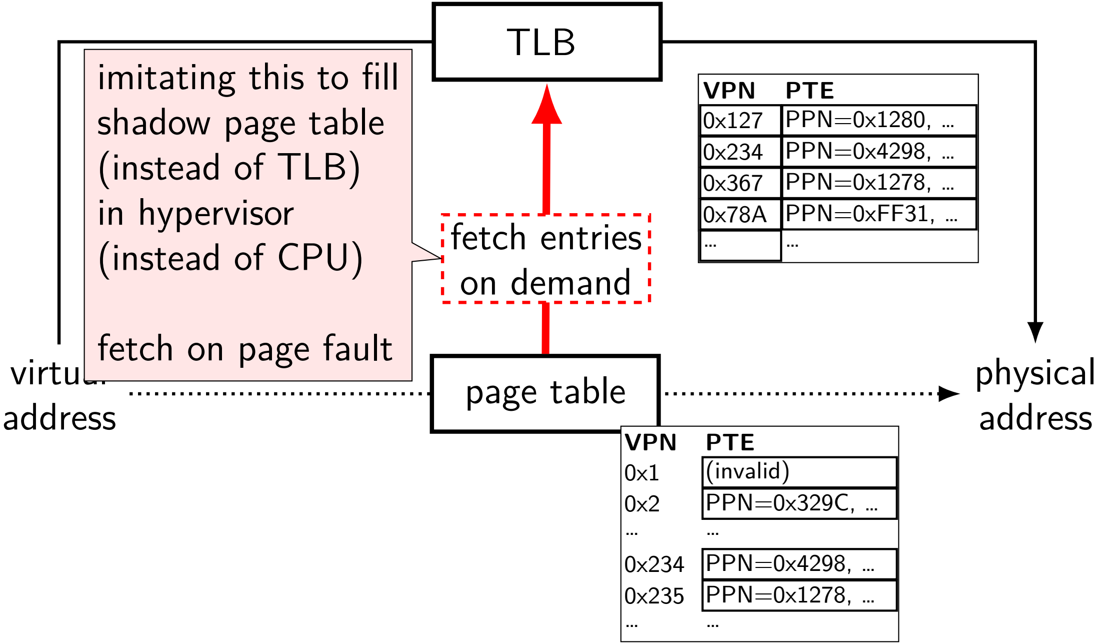
Interlude: TLB (no virtualization)



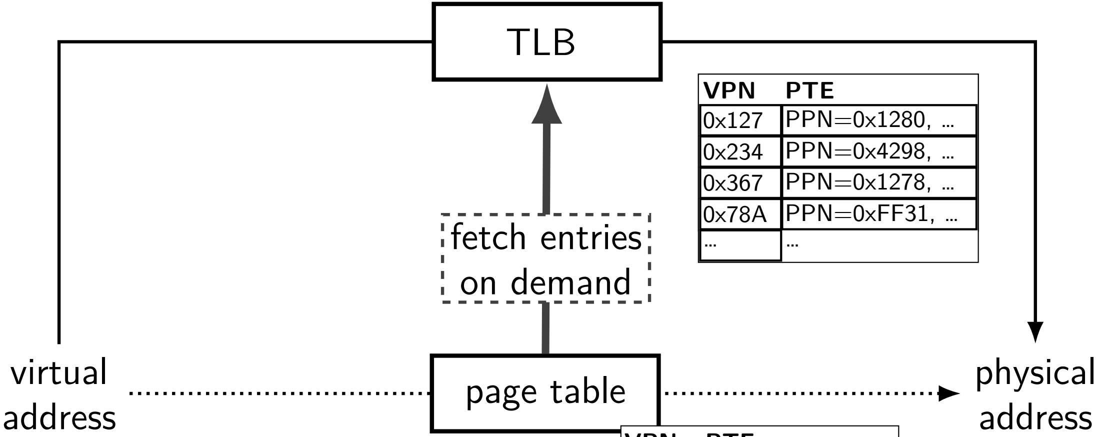
VPN	PTE
0x127	PPN=0x1280, ...
0x234	PPN=0x4298, ...
0x367	PPN=0x1278, ...
0x78A	PPN=0xFF31, ...
...	...

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

Interlude: TLB (no virtualization)



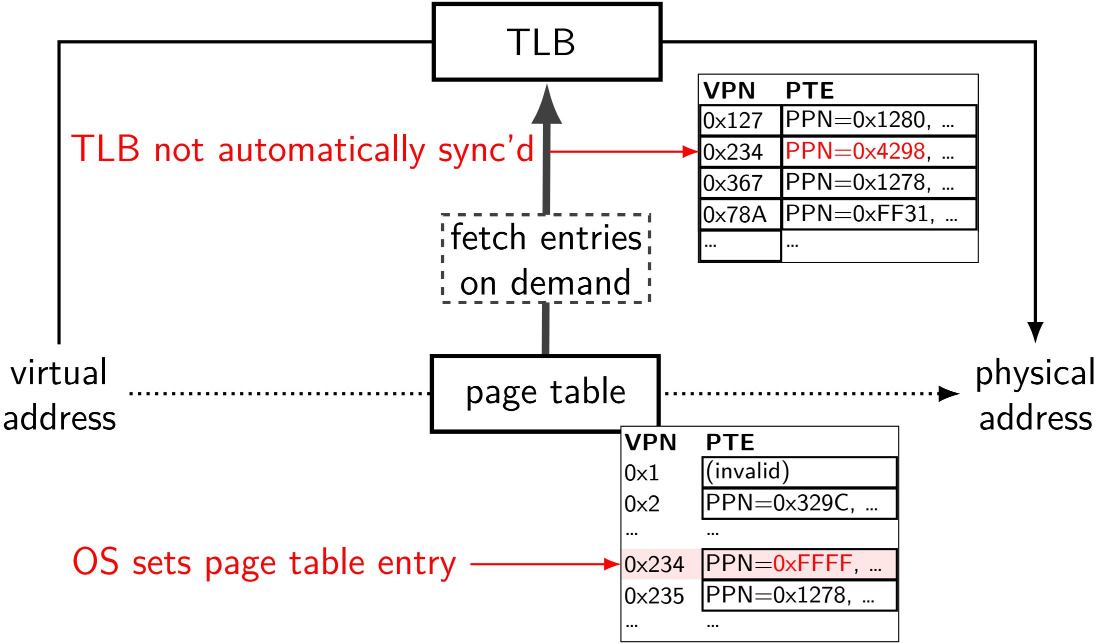
Interlude: TLB (no virtualization)



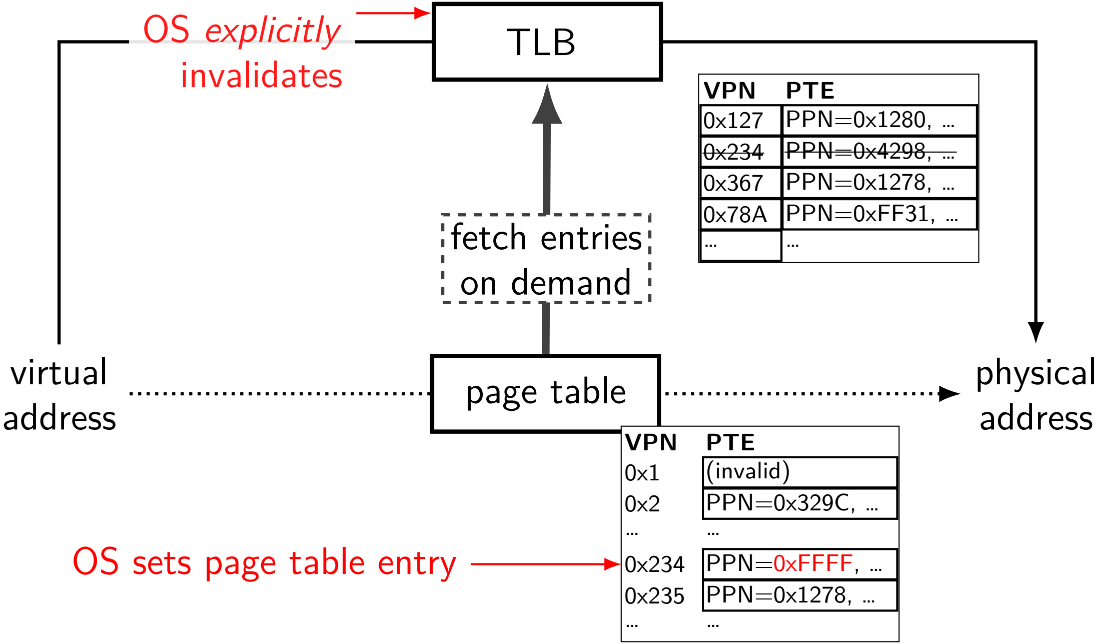
VPN	PTE
0x127	PPN=0x1280, ...
0x234	PPN=0x4298, ...
0x367	PPN=0x1278, ...
0x78A	PPN=0xFF31, ...
...	...

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

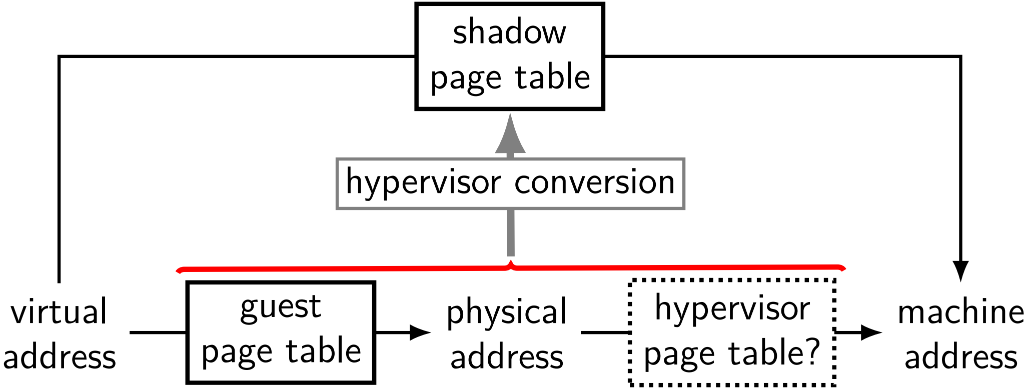
Interlude: TLB (no virtualization)



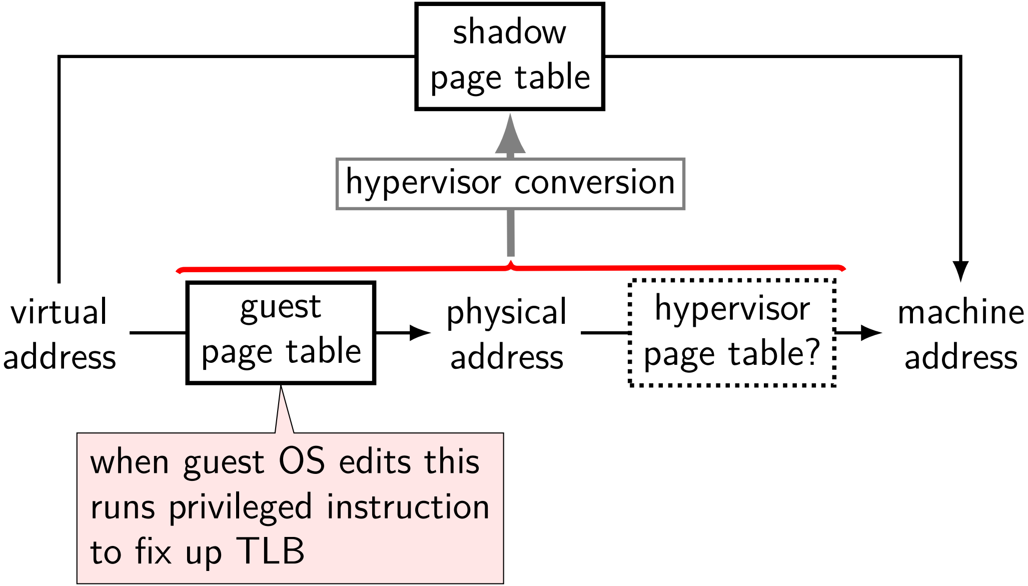
Interlude: TLB (no virtualization)



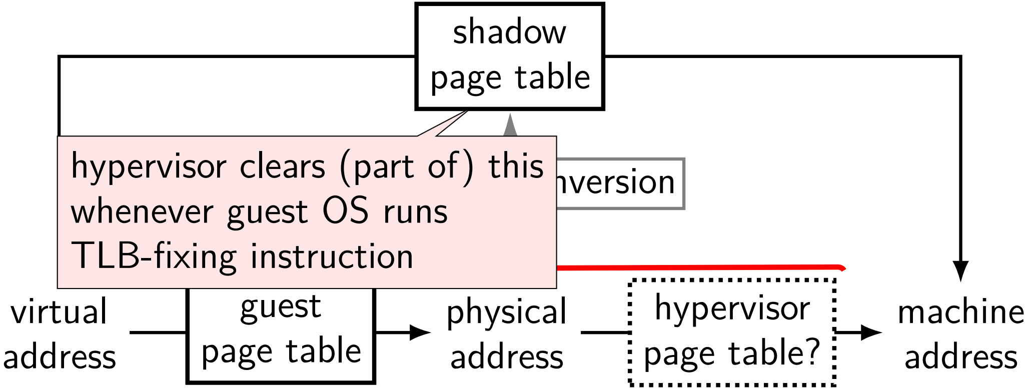
three page tables (revisited)



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

page tables and kernel mode?

guest OS can have *kernel-only* pages

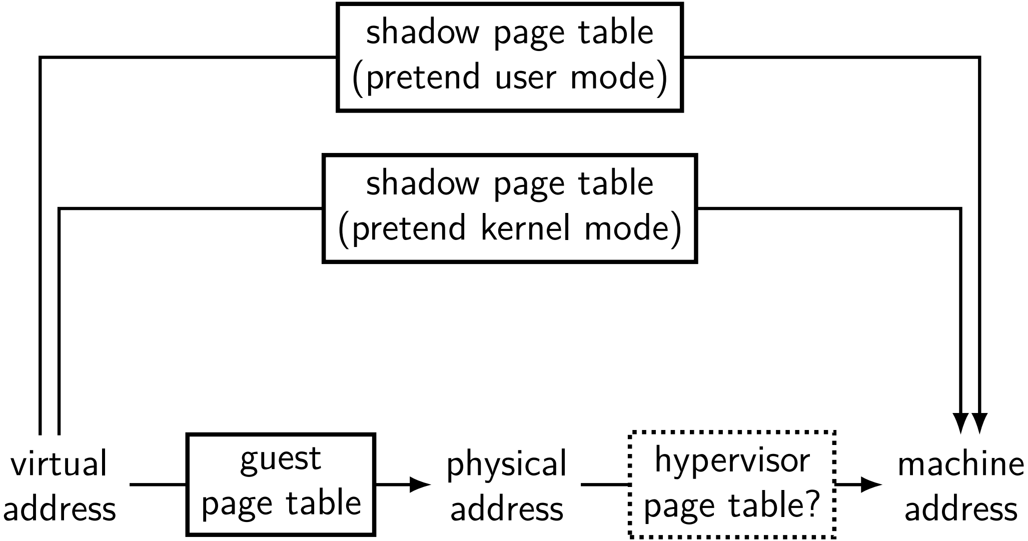
guest OS in pretend kernel mode

- 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 (or clearing)?

backup slides