

Protected Control Transfers (exception mechanisms)

- mechanism to give kernel control at a safe code location in a safe environment

Types

1. traps : 'syscall' instruction, 'int3'

- voluntarily caused by unprivileged software
 - can have a calling convention

2. interrupts : timer interrupt

- involuntarily caused by hardware
- no calling convention, so must appear transparent,
all registers must be restored

3. faults : segmentation fault

- involuntary, caused by software error*
- no calling convention, save + restore registers

* why they're caused, whether caused by unprivileged software or involuntarily, and how hardware mechanism to control transfer is restricted

PCT requirements

- cannot cause errors on its own
- configure code location / environment
- registers / hardware features define entry points into kernel
 - must be privileged
- Kernel memory protection

- changing memory layout (memory configuration) is privileged.

Protected Control Transfer Examples

sys_getpid (v-lib.h file) trap

- calls make-syscall (PID)
 - has way to move assembly to a C program
 - sets up registers / calling convention
 - i/o registers
 - input registers
 - Modifiable registers
- caller-saved
registers for syscalls* ←
- only concerns general purpose, not special purpose registers
- rax holds syscall # on input
after syscall, rax holds return value

PCT what changes?

- always changes
 - hardware saves protected values
 - ✓ .rip : runs kernel instructions
 - ✓ .cs : kernel privilege register
 - ↑ for interrupts, saved in memory
 - general purpose
 - for syscalls, swapped into other registers
 - ✓ .rflags, ✓ .rsp ← x86-64 specific

* no way to "fake" a protected control transfer

* svd0 has privilege 3 (even programs running as root don't have machine privilege, only programs in kernel do)

* modern processors have in-built support for VMs w/ multiple layers of privilege. You can always emulate privilege.

* faults can happen in kernel mode or unprivileged mode

Page Table

- provides virtual memory protection and flexible address mappings
- instructions return VIRTUAL addr $\xrightarrow[\text{mapping}]{\substack{\text{kernel} \\ \text{entered}}}$ physical address
- X86-64:
 - must be a physical address
 - %CR3 register = addr of current page table.
 - is a privileged register. stored in memory

One-level Page Table

index

0	0x3000 PTE-P
1	0xFF000 PTE-P PTE-W
2	0x7000 PTE-P PTE-W PTE-U
3	0

what physical addr does Va 0x21F6
map to? pa 0x71F6

2nd index

mem(Va, flags):
assert(flags & PTE-P);
pt = %CR3; get page table

constants, assume PTE-P |
(PTE-W, PTE-U)*

1 2

$off = (va \& 0xFFFF);$ calc offset

& every memory

$index = va >> 12;$ calc index

access requires

$t = pt[index];$ look up entry @ index

present flag

$if((e\&flags) \neq flags)$ check if entry allows

{ fault; }

mask off flags

return $(e \& \sim 0xFFFF) | off;$ add offset

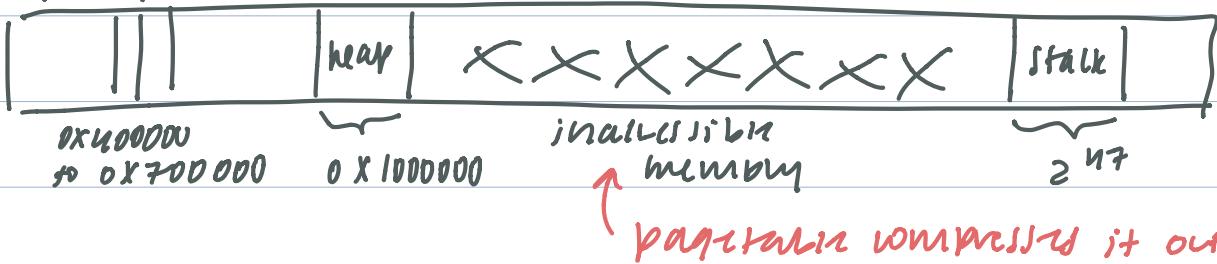
Bitwise arithmetic

• If bits = 0 $e = 0x3001$

$\& \sim 0xFFFF = 0xFFFF'FFF'FFF'FFF'FOOO$

X86-64 4-level page table

linux process addr space



9 bits/index

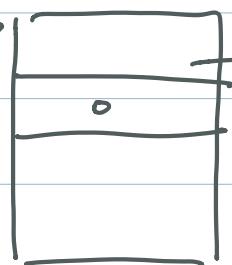
L4	L3	L2	L1	offset
47	31	21	20	11 0

va

none of the pages can be accessed by unprivileged processes

L1 page table

$\therefore CR3 \rightarrow$ L1 pagetable



L1 pagetable

2^9 indices in a pagetable

L3 page table

2^3 to store an address

$2^3 \cdot 2^9 = 2^{12} = 4096 = \text{size of pagetable}$

destination physical page w/o offset applied

