Process isolation and virtual memory

Process (two definitions):

1. An instance of a program in execution
    - e.g. running the same program twice with the same input creates two different processes
2. An abstract computer: give program illusion that all HW resources of computer are at its disposal.

Process isolation:

A process affects other processes only as explicitly permitted

Violations:

- - P modifies Q's register
- - P modifies Q's memory
- - P prevents Q from running

Implementation of process isolation:

If all processes have full privileges then it's BAD. Thus, we must have hardware support for different level of code privilege (otherwise process could always take over CPU).

- Privileged code gets full access.
- Unprivileged (process) gets partial access
- -> Kernel = Privileged code

Example:

Infinite loop attack -> avoid by turning on timer interrupt -> processes must not be able to turn off timer interrupt (so for example, processes must not be able to call "out" or "cli")

When a process violates process isolation -> abort(), error() or reboot
The kernel enforces process isolation. If a process causes abort, the kernel kills it.

Exceptional Control Flow (interrupts):

Safely run kernel after process error

- This is called exceptional control flow or interrupt

Or, safely run kernel at process request -> system call

Dangerous instruction: instruction that changes code privilege

But we can't prevent all attacks only through the CPU, example:

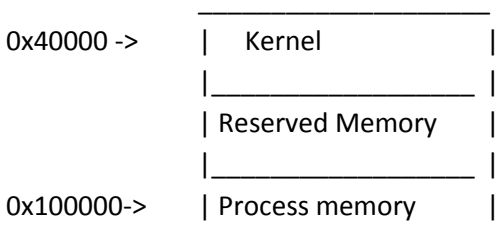
Program p-hello.c writes bytes of "while (1) ;" into the place kernel memory to where sys_getpid() redirects, then call sys_getpid() -> infinite loop attack

- Problem: memory is not isolated
- Solution: virtual memory

Virtual memory maps virtual addresses used by processes to physical memory on machine.
-> Virtual memory = Address space function AS, where:

AS(ptr, privilege_level) = Physical memory address | Fault

Picture of address space:



Do not allow the Kernel to write to the kernel memory (except for the console for memory I/O)
Memory protection problem -> CPU changes the p-hello.c writing attack to a pagefault exception, and creates a handler for these exceptions

- Normal solution for handler: kill the process
- Another possible solution: the kernel just skips the bad instruction and then the CPU continues executing the process (not used normally)
    - e.g. skip the “mov” instruction in p-hello.c by adding 7 bytes to the program counter, then restoring the registers to the CPU from memory and continuing program execution

x86 virtual memory implementation:

Two-level page table (tree of depth 2):

a Page directory: root node in tree. This is an array of 1024 pointers to pageentry_t objects (pointers have 32 bit size).
    - i.e. pageentry_t *kernel_pagedir[1024])

b Page table page: level 1 nodes. -> a pageentry_t: array of 1024 pointers into memory

Then:

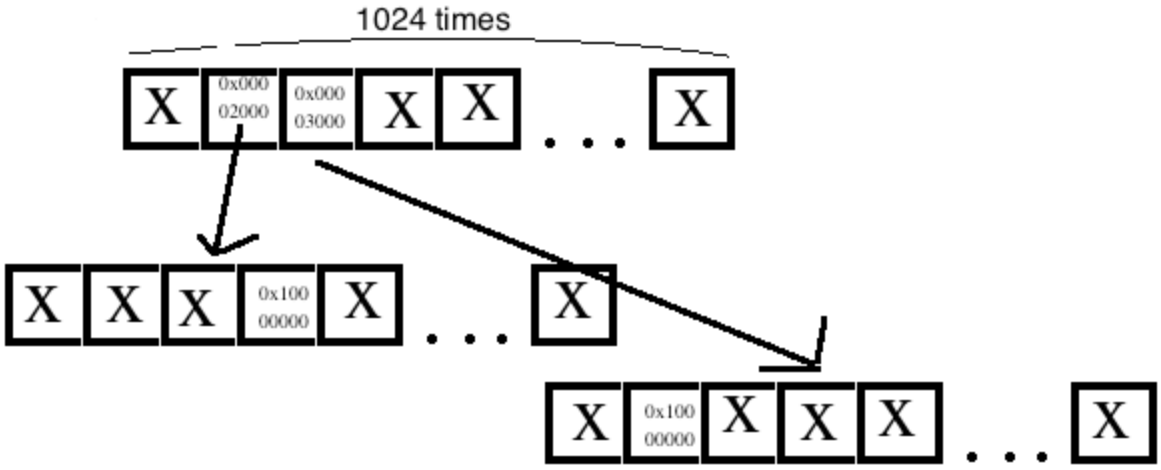
Input: virtual address (32 bit)

Page Dir: 1024 entries. Some are empty(labeled with X), some point to page table pages

Each page table page also has 1024 entries. Some are empty, some are non-empty and point to physical memory.

Output: physical address(32 bit)

Example translation with a page table:



Input value, 32 bits: PD index(22 - 31st bit) | PT index(21 - 12th bit)| offset (last 12 bits)

-> Go to physical address at the index PT index in the page table pointed to by PD index, then add the offset.

Example input value: Virtual address 0x00403005

-> binary: |0000|0000|0100|0000|0011|0000|0000|0101|

-> split into PD, PT, and offset: |0000000001|0000000011|000000000101|

-> PD index = 1; PT index = 3; offset = 5

-> PD[1] -> PT[3] gives 0x10000000. Add 5 -> 0x10000005 is the output physical memory address