3/27/24

Dual Mode Execution

OS must be able to prevent processes from doing certain things

OS: manages & abstracts hw resources

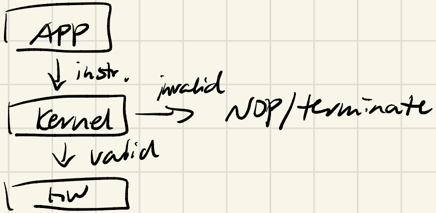
→ ease of use, common interface, managed access

How might OS achieve this?

→ option 1: inspect program binary for "bad" instr. & memory access.

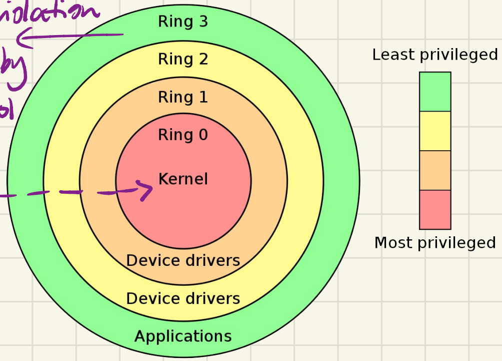
(but process may dynamically overwrite code & perform arithmetic on address to bypass the check)

→ option 2: dynamically interpose every instr. a process is executing.



highly inefficient, would be better to only involve the OS when something goes wrong

→ Protection Rings : supported by hw



privilege violation detected by hw, control transferred to kernel

Kernel sets the privilege level for each user process to be ring 3

[can you find which line - does this in xk/kernel/proc.c?]

☆ Ring 0 (kernel mode)

→ access to privileged instr.

→ eg. halt, I/O sensitive instr., update virtual memory mapping

→ access to all mapped virtual memory

Ring 1 & 2 (device drivers)

→ no access to privileged instr. but some I/O sensitive instr. (copy data from I/O port)

→ access to all mapped virtual memory

☆ Ring 3 (user mode)

→ only nonprivileged instr.

→ eg. add, push, mov, call, ret...

→ only user accessible virtual memory

Privileged access must go through the OS

-> system call : user requesting kernel services (use filesys, start new process, etc).

-> exception : hw detects privilege violation or other errors
kernel must intervene

-> interrupt : timely hw events that need to be handled by the OS

Types of Mode Transfer

→ system calls. [synchronous]

- → kernel service APIs
- → syscall, sysret instr.
- → requested by user!
- → resume on next instr. on return

→ Exceptions [synchronous]

- → unexpected problem on current instr.
  - → access invalid memory (nullptr, segfault), divide by zero, execute privileged instr.
- → terminate process, or handle the exception and resumes (retries the faulting instr.)

→ interrupts [asynchronous]

☆ needs to be handled in a timely fashion

- → hardware notifications
  - → I/O completion (disk write, packet arrival), timer interrupt
- → unrelated to the current instr.
- → resumes on the interrupted instr. on return

(resolvable) exceptions & interrupts occur in the kernel as well!

- → Who is executing in the kernel?
  - → the current process that switched into the kernel executes kernel code (handlers)

  why is this ok?

  - → upon a mode switch, hw updates process's %rip to point to kernel code. Process cannot execute arbitrary instr. in the kernel.
  - → kernel is responsible for saving & restoring process's state (+hw)