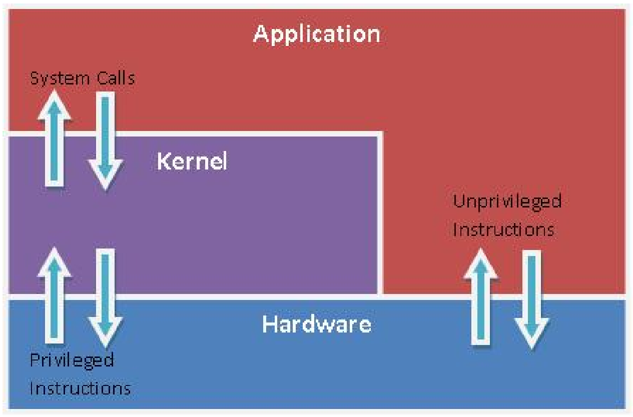
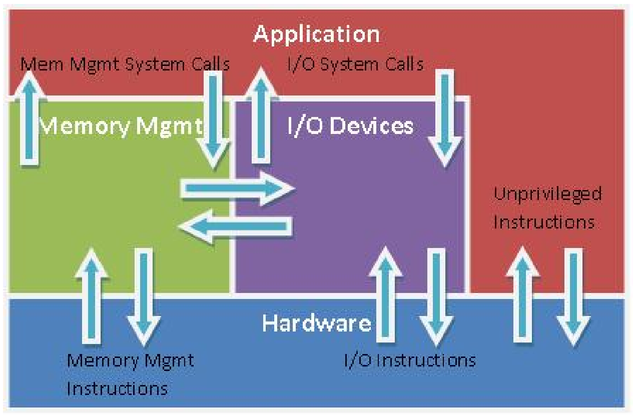
4. Virtualization



First Approach to Layered System



Second Approach to Layered System

Our model has three levels of instruction privilege:

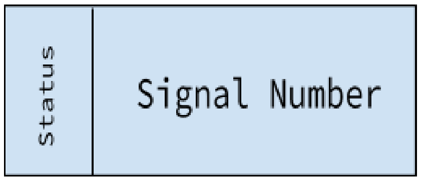
1. OS only code
    a. risky code with direct access to resources
    b. examples include inb & outb
2. Iffy Code
    a. code that accesses resourced directly some of the time
    b. movl is unprivileged unless it tries to access kernel memory
3. Unprivileged
    a. code with no access to resources

Untrusted modules can use (3) and sometimes (2)

To set up the environment for an untrusted module, the system boots in privileged mode, sets the environment for the untrusted code, and jumps to the untrusted code in unprivileged mode

But what if the untrusted application desires access to system resources
~ we send an **interrupt** with a supervisor call instruction

- if the error code matches the expected for a given operation, the privileged instruction is run!
- **Trap** jumps to kernel and raises permission
    - exit() is a trap that causes cleanup
- **Return From Trap** jumps to user & lowers permission



NOTE: for speed reasons, this exact model is not common any longer

- trap locations are defined by **<u>trap handlers</u>**
- These are held in a **<u>trap table</u>** in the kernel stack
- The key for this hash table is the **<u>system call number/interrupt number</u>**

On the hardware side, when trap occurs:

1. This information will be pushed onto the stack:
2. **eip** will be set to interrupt service routine.
3. Stack info will pop

when **reti** is called

ss	Stack Segment
esp	Stack Pointers
Flags	Processor/Privileged Flags
cs	Code Segment
eip	Instruction Pointer
Error Code	Details of trap

For interrupts, we think of a processor as a hard wired thread manager with two threads:

i) processor thread (runs scheduler)

ii) interrupt thread in kernel mode

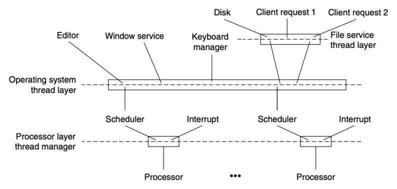


FIGURE 5.27

Thread managers applied recursively.

Each thread is built from threads of the previous layer

- We define functions that give access to privileged instructions **<u>System calls</u>**
  - this is as opposed to a **<u>Procedure call</u>**
    - a function implemented with a stack that can call other functions

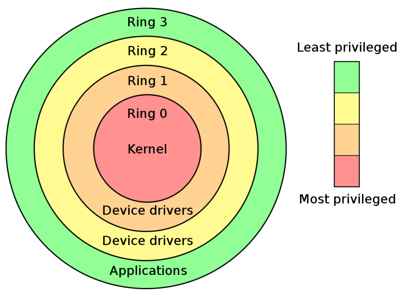
x86-64 Linux System Call Convention:

Interrupt number: %rax; arguments: %rdi, %rsi, %rdx, %r10, %r8, %r9

This is an example of <u>protected transfer of control</u>

- the system call instruction pointer is set to a location known to be sage

least privileged: display
middle privilege: devices
high privilege: memory



x86-64 uses 2 privilege bits, which lend to 4 levels of privilege

top privilege: kernel

Linux uses 2 execution modes, OSX does more

But memory is not the only thing that needs to be protected;
we must control time access to avoid stalling with bugs and time hogs!