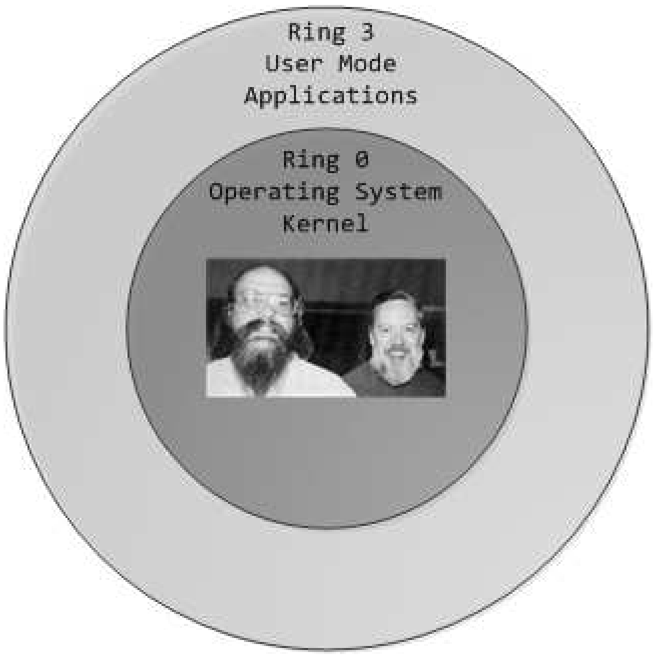
CPU Rings, Privilege, and Protection

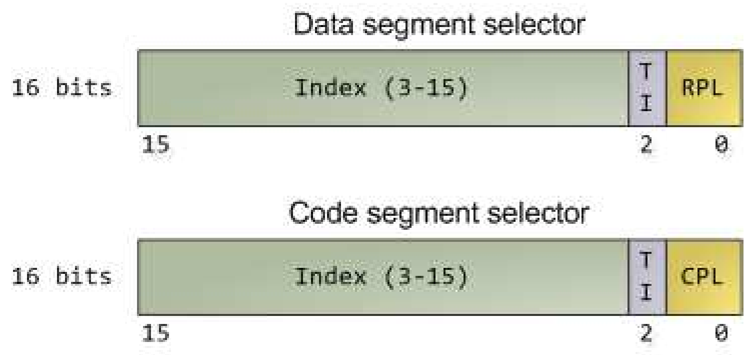
Gustavo Duarte, Aug 20th, 2008

You probably know intuitively that applications have limited powers in Intel x86 computers and that only operating system code can perform certain tasks, but do you know how this really works? This post takes a look at x86 **privilege levels**, the mechanism whereby the OS and CPU conspire to restrict what user-mode programs can do. There are four privilege levels, numbered 0 (most privileged) to 3 (least privileged), and three main resources being protected: memory, I/O ports, and the ability to execute certain machine instructions. At any given time, an x86 CPU is running in a specific privilege level, which determines what code can and cannot do. These privilege levels are often described as protection rings, with the innermost ring corresponding to highest privilege. Most modern x86 kernels use only two privilege levels, 0 and 3:



x86 Protection Rings

About 15 machine instructions, out of dozens, are restricted by the CPU to ring zero. Many others have limitations on their operands. These instructions can subvert the protection mechanism or otherwise foment chaos if allowed in user mode, so they are reserved to the kernel. An attempt to run them outside of ring zero causes a general-protection exception, like when a program uses invalid memory addresses. Likewise, access to memory and I/O ports is restricted based on privilege level. But before we look at protection mechanisms, let's see*exactly* how the CPU keeps track of the current privilege level, which involves the segment selectors from the previous post. Here they are:



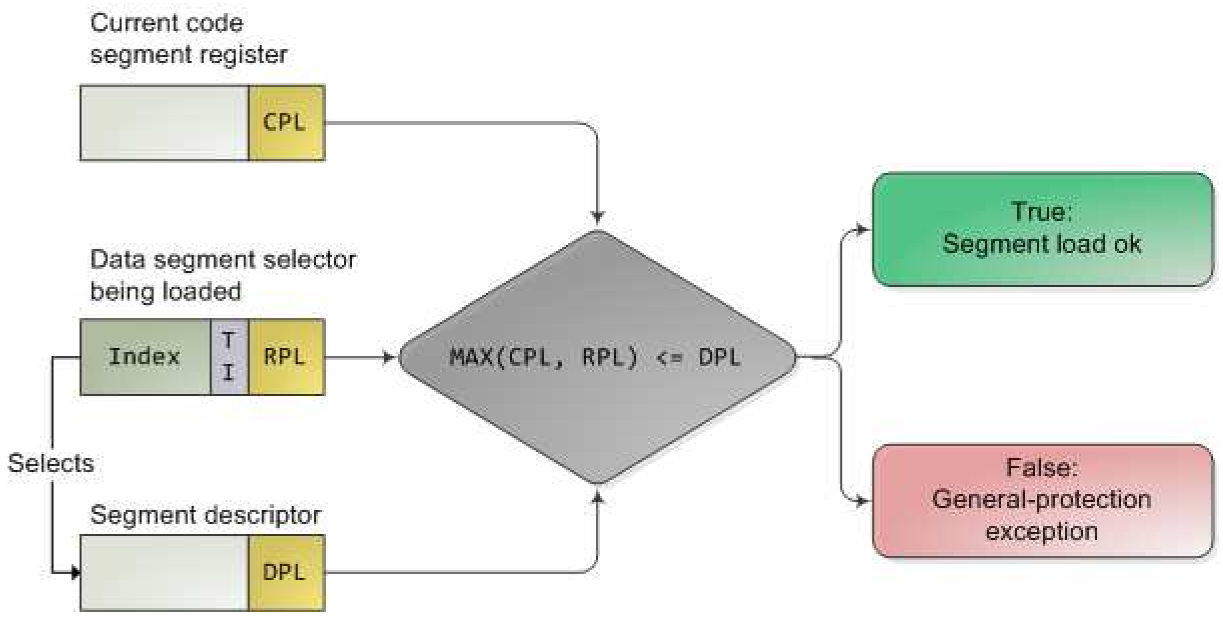
Segment Selectors – Data and Code

The full contents of data segment selectors are loaded directly by code into various segment registers such as ss (stack segment register) and ds (data segment register). This includes the contents of the Requested Privilege Level (RPL) field, whose meaning we tackle in a bit. The code segment register (cs) is, however, magical. First, its contents cannot be set directly by load instructions such as mov, but rather only by instructions that alter the flow of program execution, like call. Second, and importantly for us, instead of an RPL field that can be set by code, cs has a **Current Privilege Level** (CPL) field maintained by the CPU itself. This 2-bit CPL field in the code segment register **is always equal to** the CPU's current privilege level. The Intel docs wobble a little on this fact, and sometimes online documents confuse the issue, but that's the hard and fast rule. At any time, no matter what's going on in the CPU, a look at the CPL in cs will tell you the privilege level code is running with.

Keep in mind that the **CPU privilege level has nothing to do with operating system users**. Whether you're root, Administrator, guest, or a regular user, *it does not matter*. **All user code runs in ring 3** and **all kernel code runs in ring 0**, regardless of the OS user on whose behalf the code operates. Sometimes certain kernel tasks can be pushed to user mode, for example user-mode device drivers in Windows Vista, but these are just special processes doing a job for the kernel and can usually be killed without major consequences.

Due to restricted access to memory and I/O ports, user mode can do almost *nothing* to the outside world without calling on the kernel. It can't open files, send network packets, print to the screen, or allocate memory. User processes run in a severely limited sandbox set up by the gods of ring zero. That's why it's *impossible*, by design, for a process to leak memory beyond its existence or leave open files after it exits. All of the data structures that control such things – memory, open files, etc – cannot be touched directly by user code; once a process finishes, the sandbox is torn down by the kernel. That's why our servers can have 600 days of uptime – as long as the hardware and the kernel don't crap out, stuff can run for ever. This is also why Windows 95 / 98 crashed so much: it's not because "M$ sucks" but because important data structures were left accessible to user mode for compatibility reasons. It was probably a good trade-off at the time, albeit at high cost.

The CPU protects memory at two crucial points: when a segment selector is loaded and when a page of memory is accessed with a linear address. Protection thus mirrors memory address translation where both segmentation and paging are involved. When a data segment selector is being loaded, the check below takes place:


Current code segment register
CPL
Data segment selector being loaded
Index
T I
RPL
MAX(CPL, RPL) <= DPL
Selects
Segment descriptor
DPL
True: Segment load ok
False: General-protection exception

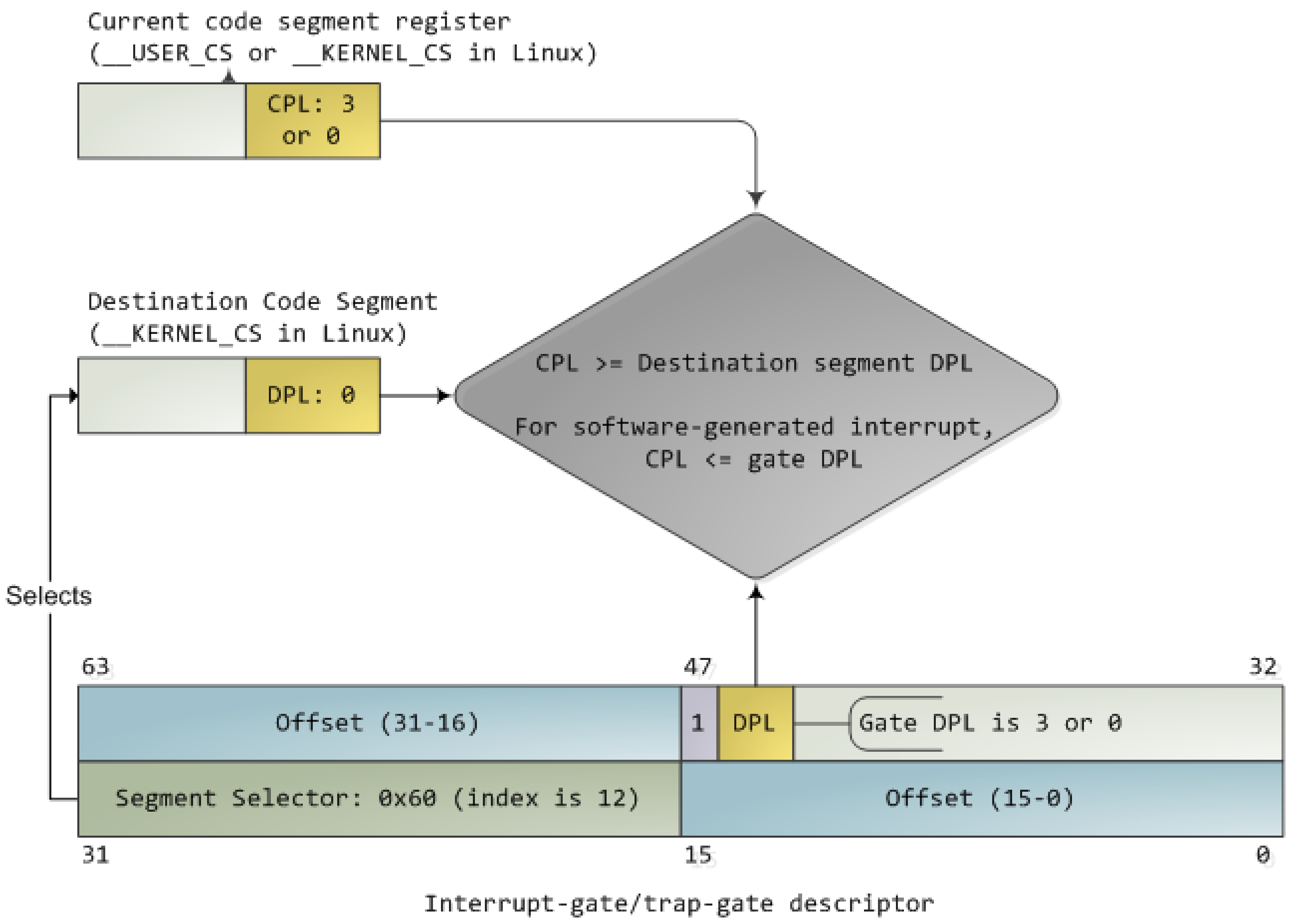
x86 Segment Protection

Since a higher number means less privilege, MAX() above picks the least privileged of CPL and RPL, and compares it to the descriptor privilege level (DPL). If the DPL is higher or equal, then access is allowed. The idea behind RPL is to allow kernel code to load a segment using lowered privilege. For example, you could use an RPL of 3 to ensure that a given operation uses segments accessible to user-mode. The exception is for the stack segment register ss, for which the three of CPL, RPL, and DPL must match exactly.

In truth, segment protection scarcely matters because modern kernels use a flat address space where the user-mode segments can reach the entire linear address space. Useful memory protection is done in the paging unit when a linear address is converted into a physical address. Each memory page is a block of bytes described by a **page table entry** containing two fields related to protection: a supervisor flag and a read/write flag. The supervisor flag is the primary x86 memory protection mechanism used by kernels. When it is on, the page cannot be accessed from ring 3. While the read/write flag isn't as important for enforcing privilege, it's still useful. When a process is loaded, pages storing binary images (code) are marked as read only, thereby catching some pointer errors if a program attempts to write to these pages. This flag is also used to implement copy on write when a process is forked in Unix. Upon forking, the parent's pages are marked read only and shared with the forked child. If either process attempts to write to the page, the processor triggers a fault and the kernel knows to duplicate the page and mark it read/write for the writing process.

Finally, we need a way for the CPU to switch between privilege levels. If ring 3 code could transfer control to arbitrary spots in the kernel, it would be easy to subvert the operating system by jumping into the wrong (right?) places. A controlled transfer is necessary. This is accomplished via **gate descriptors** and via the **sysenter** instruction. A gate descriptor is a segment descriptor of type system, and comes in four sub-types: call-gate descriptor, interrupt-gate descriptor, trap-gate descriptor, and task-gate descriptor. Call gates provide a kernel entry point that can be used with ordinary call and jmp instructions, but they aren't used much so I'll ignore them. Task gates aren't so hot either (in Linux, they are only used in double faults, which are caused by either kernel or hardware problems).

That leaves two juicier ones: interrupt and trap gates, which are used to handle hardware interrupts (*e.g.*, keyboard, timer, disks) and exceptions (*e.g.*, page faults, divide by zero). I'll refer to both as an "interrupt". These gate descriptors are stored in the **Interrupt Descriptor Table**(IDT). Each interrupt is assigned a number between 0 and 255 called a **vector**, which the processor uses as an index into the IDT when figuring out which gate descriptor to use when handling the interrupt. Interrupt and trap gates are nearly identical. Their format is shown below along with the privilege checks enforced when an interrupt happens. I filled in some values for the Linux kernel to make things concrete.


Current code segment register
(__USER_CS or __KERNEL_CS in Linux)
CPL: 3 or 0
Destination Code Segment
(__KERNEL_CS in Linux)
DPL: 0
CPL >= Destination segment DPL
For software-generated interrupt, CPL <= gate DPL
Selects
63
47
32
Offset (31-16)
1
DPL
Gate DPL is 3 or 0
Segment Selector: 0x60 (index is 12)
Offset (15-0)
31
15
0
Interrupt-gate/trap-gate descriptor

Interrupt Descriptor with Privilege Check

Both the DPL and the segment selector in the gate regulate access, while segment selector plus offset together nail down an entry point for the interrupt handler code. Kernels normally use the segment selector for the kernel code segment in these gate descriptors. An interrupt can**never** transfer control from a more-privileged to a less-privileged ring. Privilege must either stay the same (when the kernel itself is interrupted) or be elevated (when user-mode code is interrupted). In either case, the resulting CPL will be equal to to the DPL of the destination code segment; if the CPL changes, a stack switch also occurs. If an interrupt is triggered by code via an instruction like **int n**, one more check takes place: the gate DPL must be at the same or lower privilege as the CPL. This prevents user code from triggering random interrupts. If these checks fail – you guessed it – a general-protection exception happens. All Linux interrupt handlers end up running in ring zero.

During initialization, the Linux kernel first sets up an IDT in setup_idt() that ignores all interrupts. It then uses functions in include/asm-x86/desc.h to flesh out common IDT entries inarch/x86/kernel/traps_32.c. In Linux, a gate descriptor with “system” in its name is accessible from user mode and its set function uses a DPL of 3. A “system gate” is an Intel trap gate accessible to user mode. Otherwise, the terminology matches up. Hardware interrupt gates are not set here however, but instead in the appropriate drivers.

Three gates are accessible to user mode: vectors 3 and 4 are used for debugging and checking for numeric overflows, respectively. Then a system gate is set up for the SYSCALL_VECTOR, which is 0x80 for the x86 architecture. This was *the mechanism* for a process to transfer control to the kernel, to make a *system call*, and back in the day I applied for an “int 0x80” vanity license plate :). Starting with the Pentium Pro, the **sysenter** instruction was introduced as a faster way to make system calls. It relies on special-purpose CPU registers that store the code segment, entry point, and other tidbits for the kernel system call handler. When sysenter is executed the CPU does no privilege checking, going immediately into CPL 0 and loading new values into the registers for code and stack (cs, eip, ss, and esp). Only ring zero can load the sysenter setup registers, which is done in enable_sep_cpu().

Finally, when it’s time to return to ring 3, the kernel issues an **iret** or **sysexit** instruction to return from interrupts and system calls, respectively, thus leaving ring 0 and resuming execution of user code with a CPL of 3. Vim tells me I’m approaching 1,900 words, so I/O port protection is for another day. This concludes our tour of x86 rings and protection. Thanks for reading!