Segmentation and Traps

Outline

- Segmentation : Address spaces and Protection
- Trap Handling

Segmentation

Protected-mode segmentation works as follows (see handout):

- segment register holds segment selector
- selector: 13 bits of index, local vs global flag, 2-bit RPL
- selector indexes into global descriptor table (GDT)
- segment descriptor holds 32-bit base, limit, type, protection
- la = va + base ; assert(va < limit);
- choice of seg register usually implicit in instruction
  - ESP uses SS, EIP uses CS, others (mostly) use DS
  - some instructions can take far addresses:
    - `ljmp $selector, $offset`
- GDT lives in memory, CPU's GDTR register points to base of GDT
- LGDT instruction loads GDTR
- you turn on protected mode by setting PE bit in CR0 register
- What about protection?
  - instructions can only r/w/x memory reachable through seg regs
  - not before base, not after limit
  - can my program change a segment register? yes, but... to one of the permitted (accessible) descriptors in the GDT
  - can my program re-load GDTR? no!
  - how does h/w know if user or kernel?
  - Current privilege level (CPL) is in the low 2 bits of CS
  - CPL=0 is privileged O/S, CPL=3 is user
  - why can't app modify the descriptors in the GDT? it's in memory...
  - what about system calls? how do they transfer to kernel?
  - app cannot **just** lower the CPL

Traps

The x86 processor uses a table known as the *interrupt descriptor table* (IDT) to determine how to transfer control when a trap occurs. The x86 allows up to 256 different interrupt or exception entry points into the kernel, each with a different *interrupt vector*. A vector is a number between 0 and 256. An interrupt's vector is determined by the source of the interrupt: different devices, error conditions, and application requests to the kernel generate interrupts with different vectors. The CPU uses the vector as an index into the processor's IDT, which the kernel sets up in kernel-private memory of the kernel's choosing, much like the GDT. From the appropriate entry in this table the processor loads:

- the value to load into the instruction pointer (`EIP`) register, pointing to the kernel code designated to handle that type of exception.
- the value to load into the code segment (`CS`) register, which includes in bits 0-1 the privilege level at which the exception handler is to run. In PIOS, all exceptions are handled in kernel mode, privilege level 0.

Entering and Returning from Trap Handlers

When an x86 processor takes a trap while in kernel mode, it first pushes a *trap frame* onto the kernel stack, to save the old values of certain registers before the trap handling mechanism modifies them. The processor then looks up the CS and EIP of the trap handler in the IDT, and transfers control to that instruction address. The following diagram illustrates the format of the basic kernel trap frame, defining the state of the kernel stack on entry to the trap handler:

```
+--------------------+ <---- old ESP
|     old EFLAGS     |     " - 4
| 0x00000 | old CS   |     " - 8
|      old EIP       |     " - 12
+--------------------+ <---- ESP
```

For certain types of x86 exceptions, in addition to the basic three 32-bit words above, the processor pushes onto the stack another word containing an *error code*. The page fault exception, number 14, is an important example. See the x86 manuals to determine for which exception numbers the processor pushes an error code, and what the error code means in that case. When the processor pushes an error code, the stack would look as follows at the beginning of the trap handler:

The x86 processor provides a special instruction, `iret`, to return from trap handlers. It expects the kernel's stack to look like the *first* figure above, with ESP pointing to the old EIP. When the processor executes an `iret` instruction, pops the saved values of EIP, CS, and EFLAGS off the stack and back into the corresponding registers, and resumes instruction execution at the popped EIP.

Note that when returning from a trap, the processor doesn't actually know or care whether the "old" values it is popping off the stack are really the exact same values that it originally pushed onto the stack on entry to the trap handler. Think about what would happen - for better or worse - if the kernel trap handler changes these values during its execution.

The Task State Segment. The processor needs a place to save the *old* processor state before the interrupt or exception occurred, such as the original values of `EIP` and `CS` before the processor invoked the exception handler, so that the exception handler can later restore that old state and resume the interrupted code from where it left off. But this save area for the old processor state must in turn be protected from unprivileged user-mode code; otherwise buggy or malicious user code could compromise the kernel: for example, one user mode thread could change the kernel state of another thread while the latter is in a system call, or user code could simply point ESP to unmapped or read-only memory, making it impossible for the processor to push the trap frame and causing an immediate reset as described above.

For this reason, when an x86 processor takes an interrupt or trap that causes a privilege level change from user to kernel mode, it also switches to a stack in the kernel's memory. A structure called the *task state segment* (TSS) specifies the segment selector and address where this stack lives. The processor pushes (on this new stack) `SS`, `ESP`, `EFLAGS`, `CS`, `EIP`, and an optional error code. Then it loads the `CS` and `EIP` from the interrupt descriptor, and sets the `ESP` and `SS` to refer to the new stack.

Although the TSS is large and can potentially serve a variety of purposes, PIOS only uses it to define the kernel stack that the processor should switch to when it transfers from user to kernel mode. Since "kernel mode" in PIOS/Pintos/xv6 is privilege level 0 on the x86, the processor uses the `ESP0` and `SS0` fields of the TSS to define the kernel stack when entering kernel mode. PIOS/Pintos/xv6 don't use any other TSS fields.

Combined, the IDT and TSS provide the kernel with a mechanism to ensure that traps are handled only by calling well-defined entrypoints in the kernel (the interrupt vectors in the IDT) and that trap handlers will have a well-defined, protected workspace (the stack pointers in the TSS). Exactly *where* these entrypoints and kernel stacks are located is up to the kernel, however.