Traps, Handlers

Outline

- Traps : entry and return
- Handler examples

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 (and most kernels), 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:

```
+--------------------+ <---- old ESP
|     old EFLAGS     |     " - 4
| 0x00000 | old CS   |     " - 8
|      old EIP       |     " - 12
|     error code     |     " - 16
+--------------------+ <---- ESP
```

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.

Kernel Stack Management

A particularly important kernel design issue is *how many* kernel stacks there are, and which OS abstraction they are associated with. There are basically two common models:

- The xv6 kernel, like most Unix kernels, uses a *process model*: it associates a kernel stack with each user process, so that whenever the kernel is running on behalf of a particular process, it runs on that process's kernel stack, and it switches stacks whenever it switches between processes.
- The PIOS kernel, in contrast, is an *interrupt model* kernel, which means it maintains one kernel stack per *physical CPU*, irrespective of the number of processes. Kernel code running on a given CPU always runs on that CPU's permanently-assigned kernel stack regardless of what user process is running, and kernel code thus never switches kernel stacks once it has booted. The downside of this simple design is that PIOS kernel code cannot use its stack to maintain any state on behalf of processes that are not currently running: if the kernel is working on behalf of a process and needs to put the process to sleep waiting on some event, the kernel must explicitly store all relevant information about what it was doing somewhere else, such as in the process control structure, or it would be lost when the kernel switches to another process.

An Example

Let's put the above pieces together and trace through an example. Suppose the processor is executing code in user mode and encounters a divide instruction that attempts to divide by zero.

1. The processor switches to the stack defined by the `SS0` and `ESP0` fields of the TSS, which in PIOS will hold the values `CPU_GDT_KDATA` and `&cpu->kstackhi`, respectively.
2. The processor pushes the following basic trap frame onto the kernel stack, starting at `kstackhi`:

```
+--------------------+ &cpu->kstackhi
| 0x00000 | old SS   |     " - 4
|      old ESP       |     " - 8
|     old EFLAGS     |     " - 12
| 0x00000 | old CS   |     " - 16
|      old EIP       |     " - 20 <---- ESP
+--------------------+  <---- ESP
```

3. Because we're handling a divide error, which is interrupt vector 0 on the x86, the processor reads IDT entry 0 and sets `CS:EIP` to point to the handler function defined there.
4. The handler function takes control and handles the exception, for example by terminating the user environment.

As you can see, the trap frame the processor pushes on kernel entry from user is similar to the one it pushes when it is *already* in the kernel, except in this case the processor also pushes the SS and ESP registers *before* pushing the frame discussed in the preivous part of the lab. You can see this difference reflected in the comments in the definition of `trapframe` in `inc/trap.h`, and on the definitions of the `trapframe_usize()` and `trapframe_ksize()` macros in the same header file.

For those traps for which the processor also pushes an error code when taking a trap from kernel mode, as discussed in the last part, it pushes an error code in the same fashion for traps from user mode. For these traps, therefore, a trap from user mode will leave the kernel stack in the following state:

```
+--------------------+ &cpu->kstackhi
| 0x00000 | old SS   |     " - 4
|      old ESP       |     " - 8
|     old EFLAGS     |     " - 12
| 0x00000 | old CS   |     " - 16
|      old EIP       |     " - 20
|     error code     |     " - 24
+--------------------+  <---- ESP
```

Entering User Mode

What piece of register state in the processor actually defines which privilege level it is executing in at a given moment? Many architectures use a "kernel mode" flag in a control register of some kind, but x86 processors uses the low two bits of the CS register, effectively treating privilege level as a property of the currently running code segment.

We described above how the processor *leaves* user mode and enters the kernel via a trap, but how does the kernel *enter* user in the first place? Simple: the kernel "returns" to user mode - even if the processor has never been there before!

When the processor executes an `iret` instruction, it pops its standard trap frame off the stack starting with the old EIP, but it doesn't actually know or care whether *it* actually pushed that frame on the stack or if it got there some other way. Thus, the kernel can always *manufacture* a trap frame representing whatever user mode state it wants to load into the processor, and "return" from it via `iret` to enter user mode. (There are other ways to switch to user mode on the x86, but this is the most general method.)

Software Interrupts

Now that your kernel has basic exception handling capabilities and can enter user mode, you will refine it to handle traps that user mode code may cause deliberately for various purposes; we refer to such traps as *software interrupts*. There are two traps defined by the x86 processor expressly to serve as software interrupts, and PIOS defines a third one to serve as its system call mechanism:

- `T_BRKPT`: The breakpoint exception, interrupt vector 3, is normally used to allow debuggers to insert breakpoints in a program's code by temporarily replacing the relevant program instruction with the special 1-byte `int3` software interrupt instruction.
- `T_OFLOW`: The overflow exception, interrupt vector 4, allows software to generate a trap deliberately via the special `into` software interrupt instruction, if the overflow flag (FL_OF) is set when the instruction executes. The intent is for software to execute an `into` after an arithmetic operation that might overflow: if it doesn't, execution proceeds normally, but if it does, the overflow condition is immediately caught. This appealing idea has never really caught on in high-level languages, however, and is rarely used. Nevertheless, on principle we want to ensure that PIOS can handle `into` instructions in applications properly.
- `T_SYSCALL`: This value defined in `inc/trap.h` is *not* one of the 32 defined by the x86 architecture, but is somewhat arbitrarily assigned by PIOS to be 48 (0x30), one of the higher vectors in the 256-vector space. PIOS application code will invoke this trap vector deliberately with the `int` instruction when it wishes to make an explicit system call to the kernel. This is the classic way to perform system calls on the x86.

We do not want user code to be able to invoke *just any* interrupt vector in the IDT deliberately via `int` instructions, however: that might allow user code to confuse the kernel into thinking that some special event has occurred when it has not. For this reason, all IDT descriptors have a *descriptor privilege level* indicating what privilege level is required for software to invoke that interrupt vector deliberately via a software interrupt instruction. Most of these vectors are normally set to "privileged" (ring 0), but we want the vectors used for software interrupts to be set so user mode code can invoke them.