6.828 2014 Lecture 8: System calls, Interrupts, and Exceptions

- plan: user->kernel transition int instruction trap and return device interrupts
- three reasons to change from user to kernel systems calls program faults (div by zero, page fault) external device interrupts
- why do we need to take special care for user to kernel? security/isolation only kernel can touch devices, MMU, FS, other process' state, &c think of user program as a potential malicious adversary
- remember how x86 privilege levels work CPL in low 2 bits of CS CPL=0 -> can modify cr, devices, can use any PTE CPL=3 -> can't modify cr, or use devs, and PTE_U enforced
- what has to happen? save user state for future transparent resume set up for execution in kernel (stack, segments) choose a place to execute in kernel get at system call arguments do it all securely
- it is neat that interrupts, faults, system call use same mechanism the int instruction

The INT instruction

- The x86 CPU supports 256 interrupt vectors. Different hardware conditions produce interrupts through different vectors. The vector refers to an descriptor in the IDT. The CPU's IDTR register holds the (virtual) base address of the IDT. Each descriptor contains a segment selector, an offset in that segment, and a DPL.
- The INT instruction takes the following steps on a system call

(these will be similar to all interrupts and faults, though there are slight differences):

```
1. decide the vector number, in this case it's the 0x40 in int 0x40

2. fetch the interrupt descriptor for vector 0x40 from the IDT.

   the CPU finds it by taking the 0x40'th 8-byte entry starting at the physical
   address that the IDTR CPU register points to.

3. check that CPL <= DPL in the descriptor (but only if INT instruction).

4. save ESP and SS in a CPU-internal register (but only if target segment selector's PL < CPL).

5. load SS and ESP from TSS ("")

6. push user SS ("")

7. push user ESP ("")

8. push user EFLAGS

9. push user CS

10. push user EIP

11. clear some EFLAGS bits

12. set CS and EIP from IDT descriptor's segment selector and offset
```

- INT is a complex instruction. Does it really need to take all those steps? Why not let the kernel interrupt handler do some of them? For example, why does INT need to save the SS and ESP?
- xv6 set up the IDT in tvinit() and set the IDTR in idtinit(); SETGATE is in mmu.h switchuvm() specified the SS and ESP in the TSS. task segment is hardware support for multi-tasking we don't use it, other than ss0:esp this points to the kernel stack that processor switches to on interrupt/exception from user space print idt[0x40] to see how the IDT is set up to handle vector 0x40

Trap Handling

- int 0x40 entered the kernel at vector64, generated by vectors.pl. b vector64
- What is the current CPL? How was it set? Could the user abuse the INT instruction to exercise privilege or break the kernel?
- x/6x $esp in order to see what int put on the stack. Compare to handout figure? What stack is being used?
- x/3i vector64 vector64 pushes a few items on the stack and then jumps to alltraps. Why not have vector 64 in the IDT point directly to alltraps?
- Single-step alltraps until pushl %esp, then x/19x $esp. Compare with struct trapframe (x86.h)
- At the start of trap(), what is tf->trapno? How was it set?

Trap Return

- syscall() returns to trap(), and trap() returns to alltraps
- b trap.c:44 (instruction after call syscall). print *tf What is different and why? si until popal. x/19x $esp to see the trap frame again.
- single-step until iret, x/5x $esp, single-step into user space. Print the registers and stack. What will the return
- What would happen if a user program divided by zero? What if kernel code divided by zero?
- In Unix, traps often get translated into signals to the process. For example, see homework for next Wednesday Some traps, though, are (partially) handled internally by the kernel -- which ones?
- Some traps push an extra error code onto the stack (typically containing the segment descriptor that caused a fault). But this error code isn't pushed by the INT instruction. Can the user confuse the kernel by invoking INT 0xc (or any other vector that usually pushes an error code)? Why not?

Device Interrupts

- Like system calls, except: devices generate them at any time there are no arguments in CPU registers nothing to return to usually can't ignore them (unless set ignore flags in EFLAGS)
- Implementation There is hardware on the motherboard to signal the CPU when a device needs attention (e.g. the user has typed a character on the keyboard). There's usually a separate vector for each device.
- Let's look at the timer interrupt the timer hardware generates an interrupt 100 times per second allows the kernel can track the passage of time allows the kernel can time-slice among multiple running processes. The timer interrupts through vector 32.
- xv6 p idt[32], then set a breakpoint at vector32 x/20x $esp. What was the CPU doing at the time of the interrupt? What stack is being used?
- The interrupt will have pushed different numbers of words on the stack depending on whether the CPU was in user-space or the kernel; how does iret know how many words to pop?
- What prevents lots of interrupts from coming in all at once and overflowing the kernel stack? Print the registers; IF=0x200. p idt[32], p idt[64].
- trap(), when it's called for a time interrupt, does just two things
    - increment the ticks variable, and call wakeup().
    - At the end of trap, xv6 calls yield. as we will see, may cause the interrupt to return in a different process!

JOS

- JOS also has a different kernel architecture than xv6: only one kernel stack (per processor)
- The kernel is not re-entrant (cannot be interrupted), so all IDT entries are interrupt gates in JOS.