Exceptional control flow

Exceptions:
- Interrupts -> asynchronous
- Traps -> synchronous

Fault

Trap examples:
- Page faults
- Divide by 0
- Protection faults
- The int instruction

What does the system need to do on an exception?
- Save return address
- Save registers (and other state)

Jump to the handler
- Set the reason for exception
  - Privilege level
  - Set registers (Y, ESP)
- Done executing; system needs to fix up everything

How do we know where to jump?

- Interrupt vectors defined somewhere in memory

  \[ \text{IDT} \]

  \[ \text{Interrupt vector/descriptor table (IDT)} \]

  \[ \text{Code} \]

  \[ \text{Code} \]

  \[ \text{Interrupt handlers} \]

How is table initialized?

\[ \rightarrow \text{The OS: on boot OS points to handlers by writing the IDT} \]
Set privileges:

user-mode
kernel-mode (ring 0)

Privilege rings

Ring 3 is user
Ring 2
Ring 1
Ring 0

Prevents users from:

- Executing certain instructions
- Modifying certain registers
- Messing with interrupt vectors
- And lots more

Need to be in Ring 0 to run int. handler
Asking the OS to do something.

```
read(fd, buf, 100);
```

System call (syscall)

in x86:

```
movl $72, %eax
```

```
int $0x80  (128)
```

```
jmp *%eax
```

```
What happens if you get an interrupt while servicing an interrupt?
```

- Stall the new interrupt
  - easiest thing to do
  * non-reentrant code
It is common in real-time OS

- Service the second interrupt, then go back to the first Interrupt Request
- Reentrant kernel

user
int handler 1
int handler 2

→ non-reentrant

user
int handler 1
int handler 2

Reentrant kernel very hard

To prevent interrupts: interrupt enable flag (IF)