CS330: Operating Systems

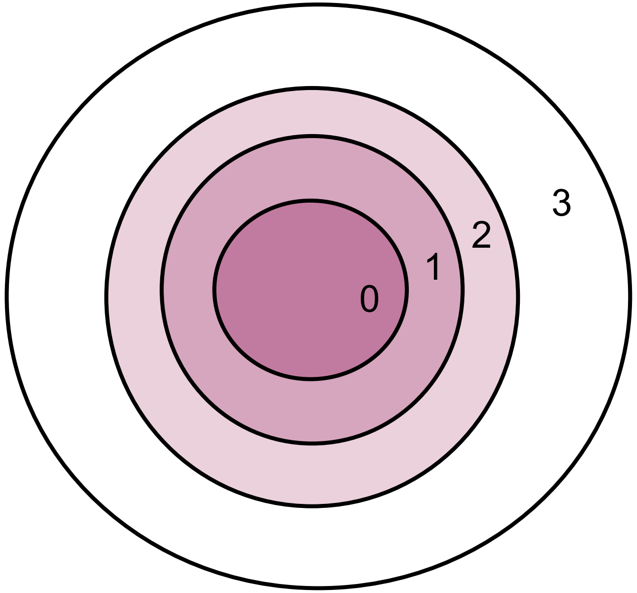
Privileged ISA (X86_64)

Recap: Limited direct execution

- Can the OS enforce limits to an executing process?
- No, the OS can not enforce limits by itself and still achieve efficiency
- OS requires support from hardware!
- What kind of support is needed from the hardware?
- CPU privilege levels: user-mode vs. kernel-mode
- Switching between modes, entry points and handlers

Today's agenda: High-level view of x86_64 support for privileged mode

X86: rings of protection



- 4 privilege levels: 0⟶ highest, 3⟶ lowest
- Some operations are allowed only in privilege level 0
- Most OSes use 0 (for kernel) and 3 (for user)
- Different kinds of privilege enforcement
  - Instruction is privileged
  - Operand is privileged

Privileged instruction: HLT (on Linux x86_64)

```
int main( )
{
  asm(“hlt;”);
}
```

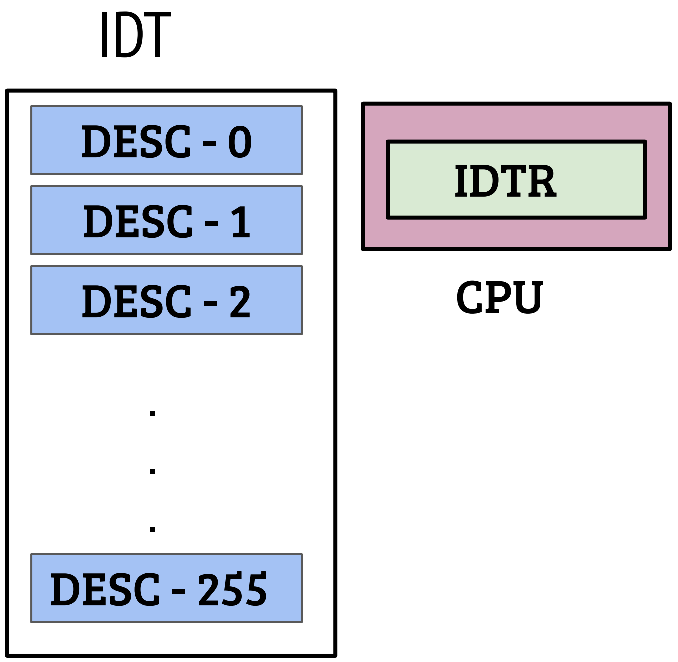
- HLT: Halt the CPU core till next external interrupt
- Executed from user space results in protection fault
- Action: Linux kernel kills the application

Privileged operation: Read CR3 (Linux x86_64)

```
#include<stdio.h>
int main( ){
  unsigned long cr3_val;
  asm volatile("mov %%cr3, %0;"
          : "=r" (cr3_val)
          :: );
 printf("%lx\n", cr3_val);
}
```

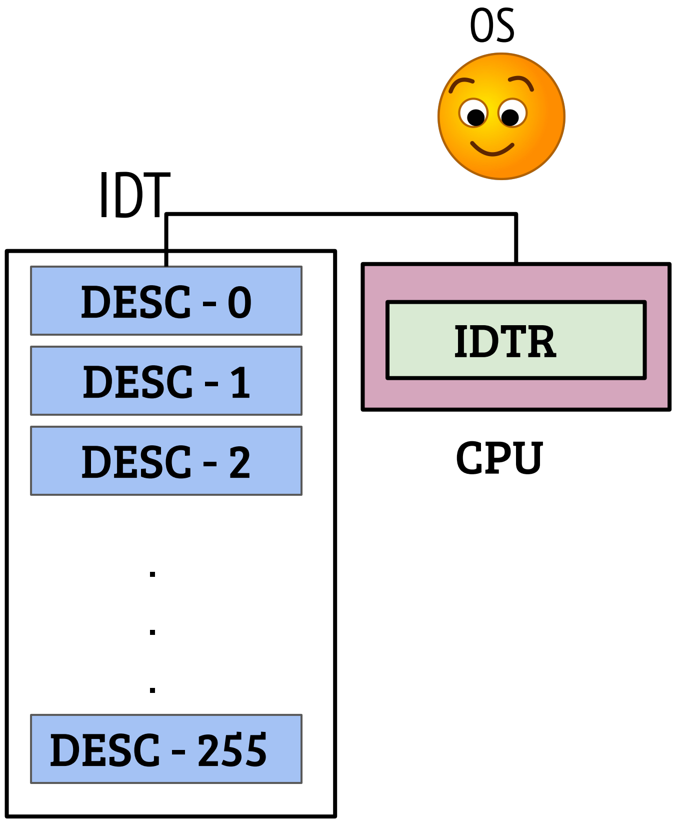
- CR3 register points to the address space translation information
- When executed from user space results in protection fault
- “mov” instruction is not privileged per se, but the operand is privileged

Interrupt Descriptor Table (IDT): gateway to handlers



- Interrupt descriptor table provides a way to define handlers for different events like external interrupts, faults and system calls by defining the descriptors
- Descriptors 0-31 are for predefined events e.g., 0 $\rightarrow$ Div-by-zero exception etc.
- Events 32-255 are user defined, can be used for h/w and s/w interrupt handling

Defining the descriptors (OS boot)



- Each descriptor contains information about handling the event
    - Privilege switch information
    - Handler address
- The OS defines the descriptors and loads the IDTR register with the address of the descriptor table (using LIDT instruction)

System call using software interrupt (INT instruction)

- INT #N: Raise a software interrupt. CPU invokes the handler defined in the IDT descriptor #N (if registered by the OS)
- Conventionally, IDT descriptor 128 (0x80) is used to define system call entry gates
- The generic system call handler invokes the appropriate handler function. How?

System call using software interrupt (INT instruction)

- INT #N: Raise a software interrupt. CPU invokes the handler defined in the IDT descriptor #N (if registered by the OS)
- Conventionally, IDT descriptor 128 (0x80) is used to define system call entry gates
- The generic system call handler invokes the appropriate handler function, how?
    - Every system call is associated with a number (defined by OS)
    - User process sends information like system call number, arguments through CPU registers which is used to invoke the actual handler

System call in Linux using INT

- Linux kernel defines system call handler for descriptor 0x80
- System call number is passed in EAX register and the return value is stored in EAX register
- Parameters are passed using the registers in the following order
    - EAX (syscall #), EBX (param #1), ECX (param #2), EDX(param #3), ESI (param #4), EDI (param #5), EBP (param #6)
- Syscall numbers can be found at /usr/include/x86_64-linux-gnu/asm/unistd_32.h