Control Transfer

1. Kernel Entry (A)
2. Thread Switch (A -> B)
3. Kernel Exit (B)
Privilege Levels

- **Ring 0**
  - Operating System Kernel
  - privileged

- **Rings 1 & 2**
  - Operating System Services
  - unprivileged

- **Ring 3**
  - User-Level Applications
  - unprivileged

- **Rings 1, 2 rarely used**
Protection Facilities

- **Segmentation**
  - Mechanism for dividing linear address space into segments
  - Different Segment Types: Code, Data, Stack, ...
  - Segment Base Address, Segment Limit
  - Mandatory mechanism, cannot be disabled

- **Paging**
  - Mechanism for translating linear to physical addresses
  - Per-page access rights (present, writable, user/superuser)
  - Implements virtual memory
  - Optional, can be turned off
Segmentation and Paging

Linear Address Space

Segmentation and Paging

Page Table Hierarchy translates Linear Address to Physical Address

If Paging is disabled, Linear Address directly maps to the Physical Address
Logical Address Translation

- Logical address consists of
  - segment selector
  - segment offset

- CPU uses table indicator (TI) in segment selector to access segment descriptor in GDT or LDT

- Segment descriptor provides segment base address, segment limit and access rights (checked by CPU)

- CPU adds segment offset to segment base address to form linear address
Segment Registers

- Processor provides 6 segment registers for holding segment selectors
  - CS (code segment)
  - DS (data segment)
  - SS (stack segment)
  - ES, FS, GS (extra data segments)

- Selector forms visible part; segment base address, limit and access rights form shadow part of the segment register (invisible)
Segment Descriptor

- **Base Addr 31..24**
- **G** Granularity
- **D** Default Size (16/32 Bit)
- **AVL** Available to Programmer
- **P** Present

- **Limit 19..16**
- **DPL** Descriptor Privilege Level
- **C** Conforming
- **R** Readable
- **A** Accessed

- **Base Addr 23..16**

- **Segment Limit 15..0**

**Code Segment Descriptor**
Flat Memory Model

- Hide segmentation mechanism by
  - setting segment base address to 0
  - setting segment size (limit) to 4 GB

- We need at least 2 segments
  - code segment
  - data/stack segment

- If kernel/user use different segment base addresses or segment limits then we need 2 segments for each
  - e.g., Small Address Spaces
Protection Checks

- CPU checks on instructions and memory references that certain protection conditions hold:
  - Segment Limit Check
  - Segment Type Check
  - Privilege Level Check
  - Restricted addressable space
  - Restricted procedure entry points
  - Restricted instruction set

- CPU raises exception if protection check fails
- Protection checks in parallel with address translation
- No big performance penalty
Privileged Instructions

- LGDT
- LLDT
- LTR
- LI DT
- MOV to %CR
- LMSW
- CLTS
- MOV to %DR

- INVD
- WBINVD
- INVLPG
- HLT
- RDMSR / WRMSR
- RDPMC / RDTSC
- CLI / STI (depending on IOPL)
Interrupts & Exceptions

- Forced transfer of execution from currently executing code to interrupt- or exception handler

- Hardware-Interrupts usually occur asynchronously in response to hardware events

- Exceptions (incl. Software Interrupts) occur synchronously while executing an instruction
Interrupt / Exception Vectors

- Vector uniquely identifies the source of the interrupt or exception (0 .. 255)
  - vectors 0 .. 31 reserved (exceptions)
  - vectors 32 .. 255 designated user vectors (interrupts)

- Interrupt Descriptor Table (IDT) is table of handler functions for all interrupts and exceptions
  - vector = offset into table
Interrupt Descriptor Table (IDT)

IDTR Register

IDT Base Address + IDT Limit → Gate for Vector n

... → Gate for Vector 3

→ Gate for Vector 2

→ Gate for Vector 1

→ Gate for Vector 0
Restricted Procedure Entry Points: Gates

- Special descriptors for protected gateways to procedures at different privilege levels
  - Subject to CPL/RPL vs. DPL checking
  - CS and EIP loaded from descriptor
  - Interrupt Gate disables interrupts, Trap Gate doesn't

- When switching privilege level, new ESP loaded from Task State Segment
  - TSS contains stack pointers for privilege levels 0,1,2
  - Kernel updates ESP0 in TSS on every thread switch to point to the current thread's kernel stack
  - Interrupt and exception handlers runs in the context and on the stack of the currently executing thread
Interrupt Gate

Selector: Segment Selector for Destination Code Segment
Offset: Offset to Procedure Entry Point in Segment
DPL: Descriptor Privilege Level (ignored for HW ints)
P: Segment Present Flag
Exception Types

- **Faults**
  - Can be corrected and allow the program to be restarted
  - Processor restores machine state prior to the execution of faulting instruction (CS and EIP point to the faulting instruction)

- **Traps**
  - Are reported immediately after execution of trapping instruction
  - Allow the program to be restarted (CS and EIP point to the instruction following the faulting instruction)

- **Aborts**
  - Are not always reported at the precise location of the exception
  - Do not allow the restart of the program
  - Usually report severe hardware errors or inconsistent state
x86 Exceptions

- 0 Divide Error (F)
- 1 Debug Exception (F/T)
- 2 NMI
- 3 Breakpoint (T)
- 4 Overflow (T)
- 5 Bound Range Exceeded (F)
- 6 Invalid Opcode (F)
- 7 Device Not Available (F)
- 8 Double Fault (A)
- 9 Coprocessor Seg. Overrun (F)
- 10 Invalid TSS (F)
- 11 Segment Not Present (F)
- 12 Stack Segment Fault (F)
- 13 General Protection Fault (F)
- 14 Page Fault (F)
- 15 reserved
- 16 FPU Math Fault (F)
- 17 Alignment Check (F)
- 18 Machine Check (A)
- 19 SIMD FP Exception (F)
Kernel Stack after Kernel Entry

- EFLAGS
- CS
- EIP
  (Error Code)

- SS
- ESP
- EFLAGS
- CS
- EIP
  (Error Code)

ESP points here

No Privileged-Level Change  Privilged-Level Change
Kernel Stack after Kernel Entry

<table>
<thead>
<tr>
<th>SS</th>
<th>ESP</th>
<th>EFLAGS</th>
<th>CS</th>
<th>EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDT_vector</td>
<td>push vector</td>
<td>entry_exc:</td>
<td>push $0</td>
<td>entry_exc_err: SAVE_STATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov %cr2, %eax</td>
<td>mov %eax, 0xc(%ebx)</td>
<td>mov %ebx, %eax</td>
</tr>
<tr>
<td></td>
<td></td>
<td>call exc_handler</td>
<td>jmp ret_from_interrupt</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAVE_STATE</td>
<td>push %ds</td>
<td>push %es</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push %fs</td>
<td>push %gs</td>
<td>pusha</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mov %esp, %ebx</td>
<td>mov $KSTCK_BEGIN, %esp</td>
<td></td>
</tr>
</tbody>
</table>
**Kernel Stack after Kernel Entry**

<table>
<thead>
<tr>
<th>Vector</th>
<th>push vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry_exc:</td>
<td>push $0</td>
</tr>
<tr>
<td>entry_exc_err:</td>
<td>SAVE_STATE</td>
</tr>
<tr>
<td>mov %cr2, %eax</td>
<td></td>
</tr>
<tr>
<td>mov %eax, 0xc(%ebx)</td>
<td></td>
</tr>
<tr>
<td>mov %ebx, %eax</td>
<td></td>
</tr>
<tr>
<td>call exc_handler</td>
<td></td>
</tr>
<tr>
<td>jmp ret_from_interrupt</td>
<td></td>
</tr>
</tbody>
</table>

SAVE_STATE

| push %ds |
| push %es |
| push %fs |
| pusha |
| mov %esp, %ebx |
| mov $KSTCK_BEGIN, %esp |
# Kernel Stack after Kernel Entry

<table>
<thead>
<tr>
<th>SS</th>
<th>IDT_vector</th>
<th>push vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP</td>
<td>entry_exc:</td>
<td>push $0</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>entry_exc_err:</td>
<td>SAVE_STATE</td>
</tr>
<tr>
<td>CS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EIP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SAVE_STATE**

```
push %ds
push %es
push %fs
push %gs
pusha
mov %esp, %ebx
mov $KSTCK_BEGIN, %esp
```
Kernel Stack after Kernel Entry

<table>
<thead>
<tr>
<th>SS</th>
<th>IDT_vector</th>
<th>push vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP</td>
<td>entry_exc:</td>
<td>push $0</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>entry_exc_err:</td>
<td>SAVE_STATE</td>
</tr>
<tr>
<td>CS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EIP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vector</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SAVE_STATE

```
push %ds
push %es
push %fs
push %gs
pusha
mov %esp, %ebx
mov $KSTCK_BEGIN, %esp
```
Kernel Stack after Kernel Entry

<table>
<thead>
<tr>
<th>SS</th>
<th>push vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP</td>
<td>push $0</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>SAVE_STATE</td>
</tr>
<tr>
<td>CS</td>
<td>mov %cr2, %eax</td>
</tr>
<tr>
<td>EIP</td>
<td>mov %eax, 0xc(%ebx)</td>
</tr>
<tr>
<td>Vector</td>
<td>mov %ebx, %eax</td>
</tr>
<tr>
<td>Error</td>
<td>call exc_handler</td>
</tr>
<tr>
<td>DS</td>
<td>jmp ret_from_interrupt</td>
</tr>
<tr>
<td>ES</td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td></td>
</tr>
<tr>
<td>GS</td>
<td></td>
</tr>
<tr>
<td>GPR (eax, ecx, edx, ebx, cr2, ...)</td>
<td>push %ds</td>
</tr>
<tr>
<td></td>
<td>push %es</td>
</tr>
<tr>
<td></td>
<td>push %fs</td>
</tr>
<tr>
<td></td>
<td>push %gs</td>
</tr>
<tr>
<td></td>
<td>pusha</td>
</tr>
<tr>
<td></td>
<td>mov %esp, %ebx</td>
</tr>
<tr>
<td></td>
<td>mov $KSTCK_BEGIN, %esp</td>
</tr>
</tbody>
</table>
Kernel Stack after Kernel Entry

<table>
<thead>
<tr>
<th>SS</th>
<th>push vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP</td>
<td>push $0</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>SAVE_STATE</td>
</tr>
<tr>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>EIP</td>
<td></td>
</tr>
<tr>
<td>Vector</td>
<td>entry_exc:</td>
</tr>
<tr>
<td>Error</td>
<td>entry_exc_err:</td>
</tr>
<tr>
<td>DS</td>
<td></td>
</tr>
<tr>
<td>ES</td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td></td>
</tr>
<tr>
<td>GS</td>
<td></td>
</tr>
<tr>
<td>GPR (eax, ecx, edx, ebx, cr2, ... )</td>
<td>SAVE_STATE</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>push %ds</td>
</tr>
<tr>
<td></td>
<td>push %es</td>
</tr>
<tr>
<td></td>
<td>push %fs</td>
</tr>
<tr>
<td></td>
<td>push %gs</td>
</tr>
<tr>
<td></td>
<td>pusha</td>
</tr>
<tr>
<td></td>
<td>mov %esp, %ebx</td>
</tr>
<tr>
<td></td>
<td>mov $KSTCK_BEGIN, %esp</td>
</tr>
</tbody>
</table>
Kernel Stack after Kernel Entry

<table>
<thead>
<tr>
<th>SS</th>
<th>IDT_vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP</td>
<td>push vector</td>
</tr>
<tr>
<td>EFLAGS</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td>entry_exc:</td>
</tr>
<tr>
<td>EIP</td>
<td>push $0</td>
</tr>
<tr>
<td>Vector</td>
<td></td>
</tr>
<tr>
<td>Error</td>
<td>entry_exc_err:</td>
</tr>
<tr>
<td>DS</td>
<td>SAVE_STATE</td>
</tr>
<tr>
<td>ES</td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td></td>
</tr>
<tr>
<td>GS</td>
<td></td>
</tr>
<tr>
<td>GPR (eax, ecx, edx, ebx, cr2, ... )</td>
<td>SAVE_STATE</td>
</tr>
</tbody>
</table>

- push %ds
- push %es
- push %fs
- push %gs
- pusha
- mov %esp, %ebx
- mov $KSTCK_BEGIN, %esp

mov %cr2, %eax
mov %eax, 0xc(%ebx)
mov %ebx, %eax
call exc_handler
jmp ret_from_interrupt
REGPARAM(1)

static void exc_handler (Exc_regs * r) {
    switch (r->vec) {
        case PAGE_FAULT:
            handle_pf (r);
            return;
        case PROTECTION_FAULT:
            handle_gp (r);
            return;
    }
}

handle_pf (Exc_regs * r) {
    mword addr = r->cr2;
    // ...
}
Kernel Exit (to Kernel Mode)

<table>
<thead>
<tr>
<th>SS</th>
<th>ESP</th>
<th>EFLAGS</th>
<th>CS</th>
<th>EIP</th>
<th>Vector</th>
<th>Error</th>
<th>DS</th>
<th>ES</th>
<th>FS</th>
<th>GS</th>
<th>GPR (eax, ecx, edx, ebx, cr2, ... )</th>
</tr>
</thead>
</table>

**IDA**

```assembly
IDT_vector
entry_exc:    push vector
entry_exc_err: push $0
SAVE_STATE
mov %cr2, %eax
mov %eax, 0xc(%ebx)
mov %ebx, %eax
call exc_handler
jmp ret_from_interrupt

ret_from_interrupt:
testb $3, 0x3c(%ebx)
jnz ret_user_iret
popa
add $24, %esp
iret
```
Kernel Exit (to Kernel Mode)

<table>
<thead>
<tr>
<th>SS</th>
<th>push vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP</td>
<td>push $0</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>SAVE_STATE</td>
</tr>
<tr>
<td>CS</td>
<td>mov %cr2, %eax</td>
</tr>
<tr>
<td>EIP</td>
<td>mov %eax, 0xc(%ebx)</td>
</tr>
<tr>
<td>Vector</td>
<td>mov %ebx, %eax</td>
</tr>
<tr>
<td>Error</td>
<td>call exc_handler</td>
</tr>
<tr>
<td>DS</td>
<td>jmp ret_from_interrupt</td>
</tr>
<tr>
<td>ES</td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td></td>
</tr>
<tr>
<td>GS</td>
<td></td>
</tr>
<tr>
<td>GPR</td>
<td></td>
</tr>
<tr>
<td>(eax, ecx, edx, ebx, cr2, ... )</td>
<td></td>
</tr>
</tbody>
</table>

ret_from_interrupt:

testb $3, 0x3c(%ebx)
jnz ret_user_iret
popa
add $24, %esp
iret
Kernel Exit (to Kernel Mode)

<table>
<thead>
<tr>
<th>SS</th>
<th>push vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP</td>
<td>entry_exc: push $0</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>entry_exc_err: SAVE_STATE</td>
</tr>
<tr>
<td>CS</td>
<td>mov %cr2, %eax</td>
</tr>
<tr>
<td>EIP</td>
<td>mov %eax, 0xc(%ebx)</td>
</tr>
<tr>
<td>Vector</td>
<td>mov %ebx, %eax</td>
</tr>
<tr>
<td>Error</td>
<td>call exc_handler</td>
</tr>
<tr>
<td>DS</td>
<td>jmp ret_from_interupt</td>
</tr>
<tr>
<td>ES</td>
<td></td>
</tr>
<tr>
<td>FS</td>
<td></td>
</tr>
<tr>
<td>GS</td>
<td></td>
</tr>
<tr>
<td>GPR</td>
<td>ret_from_interrupt:</td>
</tr>
<tr>
<td></td>
<td>testb $3, 0x3c(%ebx)</td>
</tr>
<tr>
<td></td>
<td>jnz ret_user_iret</td>
</tr>
<tr>
<td></td>
<td>popa</td>
</tr>
<tr>
<td></td>
<td>add $24, %esp</td>
</tr>
<tr>
<td></td>
<td>iret</td>
</tr>
</tbody>
</table>

(entry_exc: push $0
entry_exc_err: SAVE_STATE
mov %cr2, %eax
mov %eax, 0xc(%ebx)
mov %ebx, %eax
call exc_handler
jmp ret_from_interupt

ret_from_interupt:

| GPR (eax, ecx, edx, ebx, cr2, ... ) |
Kernel Exit (to Kernel Mode)

<table>
<thead>
<tr>
<th>SS</th>
<th>push vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry_exc:</td>
<td>push $0</td>
</tr>
<tr>
<td>entry_exc_err:</td>
<td>SAVE_STATE</td>
</tr>
</tbody>
</table>

```
entry_exc_err:
    mov %cr2, %eax
    mov %eax, 0xc(%ebx)
    mov %ebx, %eax
    call exc_handler
    jmp ret_from_interrupt
```

```
ret_from_interrupt:
    testb $3, 0x3c(%ebx)
    jnz ret_user_iret
    popa
    add $24, %esp
    iret
```
Kernel Exit (to Kernel Mode)

<table>
<thead>
<tr>
<th>SS</th>
<th>entry_exc:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESP</td>
<td>entry_exc_err:</td>
</tr>
<tr>
<td>EFLAGS</td>
<td>IDT_vector push vector</td>
</tr>
<tr>
<td>CS</td>
<td>push $0</td>
</tr>
<tr>
<td>EIP</td>
<td></td>
</tr>
</tbody>
</table>

```
entry_exc:  push $0
entry_exc_err:  SAVE_STATE
mov %cr2, %eax
mov %eax, 0xc(%ebx)
mov %ebx, %eax
call exc_handler
jmp ret_from_interrupt

ret_from_interrupt:
testb $3, 0x3c(%ebx)
jnz ret_user_iret
popa
add $24, %esp
iret
```
Kernel Exit (to Kernel Mode)

**IDT_vector**
- push vector

**entry_exc:**
- push $0

**entry_exc_err:**
- SAVE_STATE

```
mov %cr2, %eax
mov %eax, 0xc(%ebx)
mov %ebx, %eax
call exc_handler
jmp ret_from_interrupt
```

**ret_from_interrupt:**
```
testb $3, 0x3c(%ebx)
jnz ret_user_iret
popa
add $24, %esp
iret
```
Kernel Exit (to User Mode)

IDT_vector  push vector
entry_exc:  push $0
entry_exc_err:  SAVE_STATE
  mov %cr2, %eax
  mov %eax, 0xc(%ebx)
  mov %ebx, %eax
  call exc_handler
  jmp ret_from_interupt

ret_from_interrupt:
  testb $3, 0x3c(%ebx)
  jnz ret_user_iret
  popa
  add $24, %esp
  iret
Kernel Exit (to User Mode)

```c
void ret_user_iret()
{
    asm volatile ("
        lea %0, %%esp ;
        popa ;
        pop %%gs ;
        pop %%fs ;
        pop %%es ;
        pop %%ds ;
        add $8, %%esp ;
        iret ;
    " : : "m" (register_file) : "memory")
};
```
void ret_user_iret()
{
    asm volatile (" 
        "lea %0, %%esp ;" 
        "popa ;" 
        "pop %%gs ;" 
        "pop %%fs ;" 
        "pop %%es ;" 
        "pop %%ds ;" 
        "add $8, %%esp ;" 
        "iret ;" 
    : : "m" (register_file) : "memory";
}

SS
ESP
EFLAGS
CS
EIP
Vector
Error
DS
ES
FS
GS
GPR
(eax, ecx, edx, ebx, cr2, ... )
Kernel Exit (to User Mode)

```c
void ret_user_iret()
{
    asm volatile ("%0, %%esp ;"
                  "popa ;"
                  "pop %gs ;"
                  "pop %fs ;"
                  "pop %es ;"
                  "pop %ds ;"
                  "add $8, %%esp ;"
                  "iret ;"
                  : : "m" (register_file) : "memory" );
}
```
Kernel Exit (to User Mode)

```c
void ret_user_iret()
{
    asm volatile ("lea %0, %%esp ;"
                  "popa ;"
                  "pop %%gs ;"
                  "pop %%fs ;"
                  "pop %%es ;"
                  "pop %%ds ;"
                  "add $8, %%esp ;"
                  "iret ;"
                  : : "m" (register_file) : "memory")
}
```
void ret_user_iret()
{
    asm volatile (     
        "lea %0, %%esp ;"
        "popa ;"
        "popa %gs ;"
        "popa %fs ;"
        "popa %es ;"
        "popa %ds ;"
        "add $8, %%esp ;"
        "iret ;"
        : : "m" (register_file) : "memory"); 
}
void ret_user_iret()
{
    asm volatile (
        "lea %0, %%%esp ;"
        "popa ;"
        "pop %%%gs ;"
        "pop %%%fs ;"
        "pop %%%es ;"
        "pop %%%ds ;"
        "add $8, %%%esp ;"
        "iret ;"
        : : "m" (register_file) : "memory");  
}

Kernel Exit (to User Mode)
void ret_user_iret()
{
    asm volatile ("
        lea %0, %%esp ;
        popa ;
        pop %%gs ;
        pop %%fs ;
        pop %%es ;
        pop %%ds ;
        add $8, %%esp ;
        iret ;"
        : : "m" (register_file) : "memory");
}
### Kernel Exit (to User Mode)

```c
void ret_user_iret()
{
    asm volatile ("";
        "lea %0, %%esp ;"
        "popa ;"
        "pop %%gs ;"
        "pop %%fs ;"
        "pop %%es ;"
        "pop %%ds ;"
        "add $8, %%esp ;"
        "iret ;"
        : : "m" (register_file) : "memory";
    }
```
Kernel Exit (to User Mode)

```c
void ret_user_iret()
{
    asm volatile (
        "leaq 0, %es;
        "popa ;
        "pop %%gs ;
        "pop %%fs ;
        "pop %%es ;
        "pop %%ds ;
        "add $8, %esp ;
        "iret ;
        : : "m" (register_file) : "memory";
    }

```
Kernel Exit : IRET

- Load Code Segment Selector, Instruction Pointer and Flags from Stack

- Evaluate RPL of CS from the Stack
  - RPL > CPL return to other privilege level
  - otherwise return to same privilege level

- If privilege Level changes
  - Load Stack Segment Selector and Stack Pointer from Stack
  - Adjust CPL (current privilege level)
Fast Kernel Entry: SYSENTER

- Fast ring transition from any ring to ring 0

- Kernel code entry point specified via
  - SYSENTER_CS_MSR (code segment)
  - SYSENTER_EIP_MSR (entry function)
  - SYSENTER_ESP_MSR (kernel stack ptr)

- Processor...
  - disables interrupts
  - loads kernel CS (from SYSENTER_CS_MSR)
  - loads kernel SS (from SYSENTER_CS_MSR + 8)
  - loads kernel ESP (from SYSENTER_ESP_MSR)
  - loads kernel EIP (from SYSENTER_EIP_MSR)
  - does NOT save user return EIP or other user state!
Fast Kernel Exit: SYSEXIT

- Fast ring transition from any ring to ring 3

- Processor...
  - loads user CS  (from SYSENTER_CS_MSR + 16)
  - loads user SS  (from SYSENTER_CS_MSR + 24)
  - loads user ESP (from ECX)
  - loads user EIP  (from EDX)
  - enables interrupts