Microkernel Construction
Kernel Entry / Exit

Nils Asmussen

05/02/2019
Outline

- x86 Details
  - Protection Facilities
  - Interrupts and Exceptions
  - Instructions for Entry/Exit

- Entering NOVA

- Leaving NOVA
Privilege Levels

- **Ring 0**
  - Operating System Kernel
  - Privileged

- **Ring 1 & 2**
  - Operating System Services
  - Unprivileged

- **Ring 3**
  - User-level Applications
  - Unprivileged
Protection Facilities

**Segmentation**
- Mechanism for dividing linear address space into segments
- Different segment types: code, data, stack, ...
- Segment has base address and limit
- Mandatory mechanism; cannot be disabled

**Paging**
- Mechanism for translating linear to physical addresses
- Splits virt. & phys. address space into same-size pages
- Per-page access rights (present, writable, user/kernel)
- Optional; can be turned off (not on x86_64)
Logical address translation involves the following steps:

- **Logical address consists of**
  - Segment selector
  - Segment offset

- **CPU uses table indicator in segment selector to access descriptor in GDT/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**
Logical Address Translation

Logical Address:

- Seg. Selector
- Seg. Offset

GDT
Linear Address Space

Linear Address:

- Seg. Desc
- Limit
- Base Address

Linear Address Space

Segment

6 / 45
Segment Registers

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

- Selector forms visible part

- Base address, limit and access rights form shadow part
Segment Descriptor

Figure: Code Segment Descriptor

G  Granularity  DPL  Descriptor Privilege Level
D  Default Size (16/32 Bit)  C  Conforming
AVL  Available to Programmer  R  Readable
P  Present  A  Accessed
Flat Memory Model

- Hide segmentation mechanism by
  - Setting segment base address to 0
  - Setting segment limit to 4 GiB
- Mandatory for x86_64
- We need at least 2 segments
  - Code segment
  - Data/Stack segment
- If kernel/user use different segment base addresses or segment limits, we need 2 additional segments for each
CPU checks on instructions and memory references that certain protection conditions hold:

- Segment limit check
- Segment type check
- Privilege level check
- 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
- LIDT
- MOV to CR
- LMSW
- CLTS
- MOV to DR
- INVD
- WBINVD
- INVLPG
- HLT
- RDMSR / WRMSR
- RDPMC / RDTSC
- CLI / STI
Interrupts & Exceptions

- Forced transfer of execution from currently executing code to interrupt- or exception handler.
- Hardware interrupts occur asynchronously in response to hardware events.
- Exceptions (including software interrupts) occur synchronously to the instruction stream.
Vector uniquely identifies source of the interrupt or exception
- Vectors 0 .. 31 are used for exceptions
- Vectors 32 .. 255 are designated user vectors (e.g., interrupts)

Interrupt Descriptor Table (IDT) is table of handler functions for all interrupts and exceptions
- Setup by OS
- Hardware is informed about it via LIDT instruction
- Vector = offset into IDT
Interrupt Descriptor Table (IDT)

IDTR register:

IDT base address | IDT limit

+ 

Gate for Vector 0
Gate for Vector 1
Gate for Vector 2
...
Gate for Vector n

...
Descriptors to call 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 levels, new ESP loaded from TSS
- TSS contains stack pointers for privilege levels 0, 1 and 2
- Kernel updates SP_0 in TSS on every thread switch
- Interrupt and exception handlers runs current thread’s context
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 resumed
  - Processor restores machine state prior to the execution of faulting instruction (CS and EIP point to faulting instruction)

- **Traps**
  - Reported immediately after execution of trapping instruction
  - Allow the program to be resumed (CS and EIP point to following instruction)

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

- 0: Divide Error (F)
- 1: Debug Exception (F/T)
- 2: Non-maskable Interrupt
- 3: Breakpoint (T)
- 4: Overflow (T)
- 5: Bound Range Exc. (F)
- 6: Invalid Opcode (F)
- 7: Device Not Available (F)
- 8: Double Fault (A)
- 9: Coproc. Seg. Overr. (F)
- 10: Invalid TSS (F)
- 11: Segm. Not Present (F)
- 12: Stack Segment Fault (F)
- 13: General Prot. Fault (F)
- 14: Pagefault (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

No Priv. Level Change

Priv. Level Change
### Kernel Entry

<table>
<thead>
<tr>
<th>SS</th>
<th>ESP</th>
<th>EFLAGS</th>
<th>CS</th>
<th>EIP</th>
</tr>
</thead>
</table>

```assembly
entry_6:
    push $6
    jmp entry_exc

entry_exc:
    push $0

entry_exc_err:
    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
```
entry_6:
  push $6
  jmp entry_exc

entry_exc:
  push $0

entry_exc_err:
  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
entry_6:
  push $6
  jmp entry_exc

entry_exc:
  push $0

entry_exc_err:
  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
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></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Vector=6  
Error=0  

DS  
ES  
FS  
GS

entry_6:  
push $6  
jmp entry_exc

text_6:  
push $0  
entry_exc:  
push %dx  
push %es  
push %fs  
push %gs  
pusha  

mov %esp, %ebx  
mov $KSTCK_BEGIN, %esp  
mov %cr2, %eax  
mov %eax, Ox(N%ebx)  
mov %ebx, %eax  
call exc_handler  
jmp ret_from_interrupt
Kernel Entry

SS
ESP
EFLAGS
CS
EIP
Vector=6
Error=0
DS
ES
FS
GS
GPR
(eax, ecx, edx, ebx, cr2, ...)

entry_6:
  push $6
  jmp entry_exc

entry_exc:
  push $0

entry_exc_err:
  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
entry_6:  
  push $6  
  jmp entry_exc

entry_exc:  
  push $0

entry_exc_err:  
  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
<table>
<thead>
<tr>
<th>SS</th>
<th>ESP</th>
<th>EFLAGS</th>
<th>CS</th>
<th>EIP</th>
<th>Vector=6</th>
<th>Error=0</th>
<th>DS</th>
<th>ES</th>
<th>FS</th>
<th>GS</th>
<th>GPR (eax, ecx, edx, ebx, cr2, ...)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>entry_6: push $6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>jmp entry_exc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>entry_exc: push $0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>entry_exc_err: push %ds</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>push %es</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>push %fs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>push %gs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>pusha</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mov %esp, %ebx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mov $KSTCK_BEGIN, %esp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mov %cr2, %eax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mov %eax, 0xc(%ebx)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>mov %ebx, %eax</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>call exc_handler</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>jmp ret_from_interrupt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

26 / 45
REGPARM(1) void exc_handler (Exc_regs *r) {
    switch (r->vec) {
    case Cpu::EXC_TS:
        handle_exc_ts (r);
        break;
    case Cpu::EXC_PF:
        handle_exc_pf (r);
        break;
    ...
    }
}

void handle_exc_pf (Exc_regs *r) {
    mword addr = r->cr2;
    if(Pd::current->Space_mem::loc[Cpu::id].sync_from (Pd::current->Space_mem::hpt, addr))
        return;
    ...
}
Kernel Exit to Kernel Mode

| EFLAGS | CS | EIP | Vector=6 | Error=0 | DS | ES | FS | GS | GPR (eax, ecx, edx, ebx, cr2, ...) |

...  

```
mov %esp, %ebx
mov $KSTCK_BEGIN, %esp
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

| EFLAGS | CS | EIP | Vector=6 | Error=0 | DS | ES | FS | GS | GPR (eax, ecx, edx, ebx, cr2, ...) |

...  

```c
mov %esp, %ebx
mov $KSTCK_BEGIN, %esp
mov %cr2, %eax
mov %eax, 0xc(%ebx)
mov %ebx, %eax
call exc_handler
```

```c
jmp ret_from_interrupt
```

```c
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>EFLAGS</th>
<th>CS</th>
<th>EIP</th>
<th>Vector=6</th>
<th>Error=0</th>
<th>DS</th>
<th>ES</th>
<th>FS</th>
<th>GS</th>
<th>GPR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(eax, ecx, edx, ebx, cr2, ...)</td>
</tr>
</tbody>
</table>

... 

mov %esp, %ebx
mov $KSTCK_BEGIN, %esp
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>EFLAGS</th>
<th>CS</th>
<th>EIP</th>
<th>Vector=6</th>
<th>Error=0</th>
<th>DS</th>
<th>ES</th>
<th>FS</th>
<th>GS</th>
</tr>
</thead>
</table>

... 

mov %esp, %ebx
mov $KSTCK_BEGIN, %esp
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
...  
mov %esp, %ebx  
mov $KSTCK_BEGIN, %esp  
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
... 

```
mov %esp, %ebx
mov $KSTCK_BEGIN, %esp
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

<table>
<thead>
<tr>
<th>SS</th>
<th>ESP</th>
<th>EFLAGS</th>
<th>CS</th>
<th>EIP</th>
<th>Vector=6</th>
<th>Error=0</th>
<th>DS</th>
<th>ES</th>
<th>FS</th>
<th>GS</th>
<th>GPR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(eax, ecx,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>edx, ebx,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>cr2,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

...  

```assembly
mov %esp, %ebx
mov $KSTCK_BEGIN, %esp
mov %cr2, %eax
mov %eax, 0xc(%ebx)
mov %ebx, %eax
call exc_handler

jmp ret_from_interrupt
```

**ret_from_interrupt:**

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

<table>
<thead>
<tr>
<th>SS</th>
<th>ESP</th>
<th>EFLAGS</th>
<th>CS</th>
<th>EIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector=6</td>
<td>Error=0</td>
<td>DS</td>
<td>ES</td>
<td>FS</td>
</tr>
<tr>
<td>GS</td>
<td>GPR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(eax, ecx, edx, ebx, cr2, ...)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
void ret_user_iret() {
    handle_hazards();

    asm volatile (        
        "lea %0, %%esp;"
        "popa;"
        "pop %gs;"
        "pop %fs;"
        "pop %es;"
        "pop %ds;"
        "add $8, %%esp;"
        "iret;"
        : : "m" (current->regs) : "memory"
    );
    UNREACHED;
}
```
```
void ret_user_iret() {
    handle_hazards();

    asm volatile (  
        "lea %0, %%esp;"  
        "popa;"  
        "pop %gs;"  
        "pop %fs;"  
        "pop %es;"  
        "pop %ds;"  
        "add $8, %%esp;"  
        "iret;"
    : : "m" (current->regs) : "memory"
    );
    UNREACHED;
}
```
### Kernel Exit to User Mode

<table>
<thead>
<tr>
<th>SS</th>
<th>ESP</th>
<th>EFLAGS</th>
<th>CS</th>
<th>EIP</th>
<th>Vector=6</th>
<th>Error=0</th>
<th>DS</th>
<th>ES</th>
<th>FS</th>
<th>GS</th>
<th>GPR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(eax, ecx, edx, ebx</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>, cr2, ... )</td>
</tr>
</tbody>
</table>

```c
void ret_user_iret() {
    handle_hazards();

    asm volatile(
        "lea %0, %%esp;"
        "popa;"
        "pop %%gs;"
        "pop %%fs;"
        "pop %%es;"
        "pop %%ds;"
        "add $8, %%esp;"
        "iret;"
    : : "m" (current->regs) : "memory"
    );
    UNREACHED;
}
```
void ret_user_iret() {
    handle_hazards();

    asm volatile (  
        "lea %0, %%esp;"  
        "popa;"  
        "pop %gs;"  
        "pop %fs;"  
        "pop %es;"  
        "pop %ds;"  
        "add $8, %%esp;"  
        "iret;"
        : : "m" (current->regs) : "memory"
    );
    UNREACHED;
}
void ret_user_iret() {
    handle_hazards();

    asm volatile ("lea %0, %%esp;"
                  "popa;"
                  "pop %%gs;"
                  "pop %%fs;"
                  "pop %%es;"
                  "pop %%ds;"
                  "add $8, %%esp;"
                  "iret;"
    : : "m" (current->regs) : "memory"
    );
    UNREACHED;
}
### Kernel Exit to User Mode

<table>
<thead>
<tr>
<th>SS</th>
<th>ESP</th>
<th>EFLAGS</th>
<th>CS</th>
<th>EIP</th>
</tr>
</thead>
</table>

```c
void ret_user_iret() {
    handle_hazards();

    asm volatile (
        "lea %0, %%esp;"
        "popa;"
        "pop %gs;"
        "pop %fs;"
        "pop %es;"
        "pop %ds;"
        "add $8, %%esp;"
        "iret;"
        : : "m" (current->regs) : "memory"
    );
    UNREACHED;
}
```
void ret_user_iret() {
    handle_hazards();

    asm volatile ( 
        "lea %0, %%esp;"
        "popa;"
        "pop %%gs;"
        "pop %%fs;"
        "pop %%es;"
        "pop %%es;"
        "pop %%ds;"
        "add $8, %%esp;"
        "iret;"
        : : "m" (current->regs) : "memory"
    );
    UNREACHED;
}

- Load code segment selector, EIP and flags from stack
- Evaluate RPL of CS from stack
  - RPL > CPL: return to other privilege level
  - otherwise: return to same privilege level
- If privilege level changes
  - Load stack segment selector and ESP from stack
  - Adjust CPL
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 pointer)
- CPU ...
  - 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 ring transition from any ring to ring 3

CPU ...
- 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
Next Weeks

- May, 9th: Exercise 1 (Kernel entry/exit) in **room E046**
- May, 16th: Interprocess communication
- May, 23rd: Exercise 2 (Multiboot, ELF) in **room E046**
- May, 30th: – Himmelfahrt –
- June, 6th: Exercise 3 (Thread switching) in **room E046**
- June, 13th: – Pfingsten –
- June, 20th: – OUTPUT –
- …