



操作系统

Operating System

Lec4 实验一 bootloader启动ucore os
清华大学计算机系

大 纲

- x86 启动顺序
- C函数调用
- gcc内联汇编 (inline assembly)
- x86-32下的中断处理

- 理解x86-32平台的启动过程
- 理解x86-32的实模式、保护模式
- 理解段机制

x86启动顺序

x86启动顺序 – 寄存器初始值

Table 9-1. IA-32 Processor States Following Power-up, Reset, or INIT

Register	Pentium 4 and Intel Xeon Processor	P6 Family Processor (Including DisplayFamily = 06H)	Pentium Processor
EFLAGS ¹	00000002H	00000002H	00000002H
EIP	0000FFF0H	0000FFF0H	0000FFF0H
CR0	60000010H ²	60000010H ²	60000010H ²
CR2, CR3, CR4	00000000H	00000000H	00000000H
CS	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = F000H Base = FFFF0000H Limit = FFFFH AR = Present, R/W, Accessed
SS, DS, ES, FS, GS	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed	Selector = 0000H Base = 00000000H Limit = FFFFH AR = Present, R/W, Accessed
EDX	00000FxxH	000n06xxH ³	000005xxH
EAX	0 ⁴	0 ⁴	0 ⁴
EBX, ECX, ESI, EDI, EBP, ESP	00000000H	00000000H	00000000H

摘自"IA-32 Intel体系结构软件开发手册"

x86启动顺序 – 第一条指令

- $CS = F000H$, $EIP = 0000FFF0H$
- 实际地址是:
 $Base + EIP = FFFF0000H + 0000FFF0H = FFFFFFF0H$
这是BIOS的EPROM (Erasable Programmable Read Only Memory) 所在地
- 当CS被新值加载, 则地址转换规则将开始起作用
- 通常第一条指令是一条长跳转指令(这样CS和EIP都会更新)到BIOS代码中执行

x86启动顺序 – 处于实模式的段

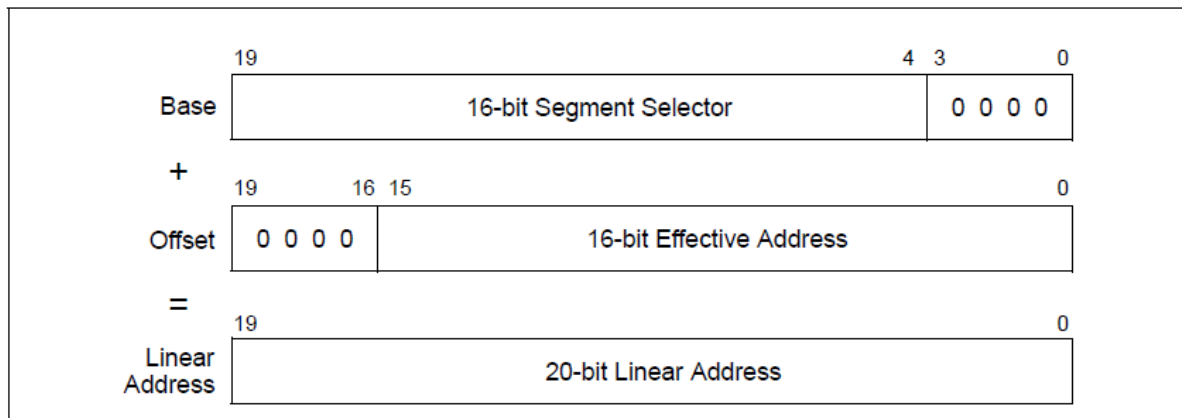


Figure 20-1. Real-Address Mode Address Translation

摘自"IA-32 Intel体系结构软件开发手册"

- 段选择子 (Segment Selector) : CS, DS, SS, ...
- 偏移量 (Offset) : EIP

x86启动顺序 – 从BIOS到Bootloader

- BIOS 加载存储设备（比如软盘、硬盘、光盘、USB盘）上的第一个扇区(主引导扇区，Master Boot Record, or MBR) 的512字节到内存的 0x7c00 ...
- 然后转跳到 @ 0x7c00的第一条指令开始执行

x86启动顺序 – 从bootloader到OS

■ bootloader做的事情:

- ▶ 使能保护模式 (protection mode) & 段机制 (segment-level protection)
- ▶ 从硬盘上读取kernel in ELF 格式的ucore kernel (跟在MBR后面的扇区)并放到内存中固定位置
- ▶ 跳转到ucore OS的入口点 (entry point) 执行 , 这时控制权到了ucore OS中

x86启动顺序 – 实模式 (history)

型号	位宽	总线位宽	地址位	寻址空间	备注
4004	4	4		640B	1971-11-15
8008	8	8		16K	1972-4-1
8080	8	8	16	64K	1974-4-1
8086	16	16	20	1M	1978-6-8 X86 起源
8088	除了外部总线位宽是 8 位，其他参数和 8086 完全一致				
80186	16	16	20	1M	简单认为较 8086 多了几条指令而已
80286	16	16	24	16M	多任务、多用户系统的核心
80386	32	32	32	4G	包含了分页的虚拟内存机制
80486	32	32	32	4G	
Pentium 1,2,3,4,酷睿, 酷睿 2					

x86启动顺序 – 实模式 (history)

段基地址:0x1234

段内偏移:0x4321

	0x	1	2	3	4	0
+	0x		4	3	2	1

0x 1 6 6 6 1

物理地址:0x16661

这种方式可以寻址的最高地址为0xFFFF:0xFFFF，其地址空间为0x00000~0x10FFEF，因为8086的地址总线是20位，最大只能访问到1MB的物理地址空间，即物理地址空间是0x00000~0xFFFFF。

8086->>80186->>80286->>80386

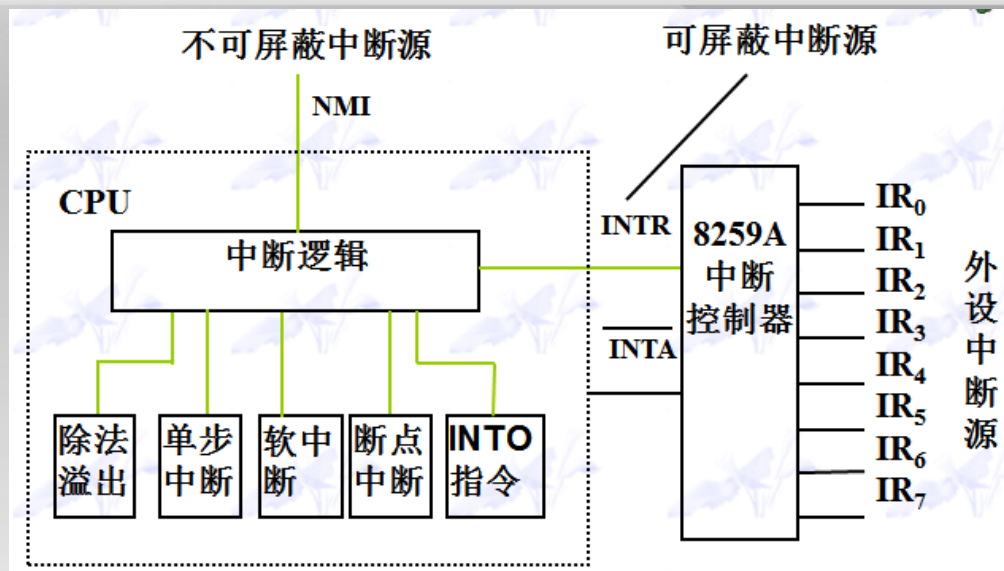
x86启动顺序 – 实模式 (history)

段基地址: 0x1234

段内偏移: 0x4321

	0x	1	2	3	4	0
+	0x		4	3	2	1
<hr/>						
	0x	1	6	6	6	1

物理地址: 0x16661



1. 中断请求
2. 中断响应

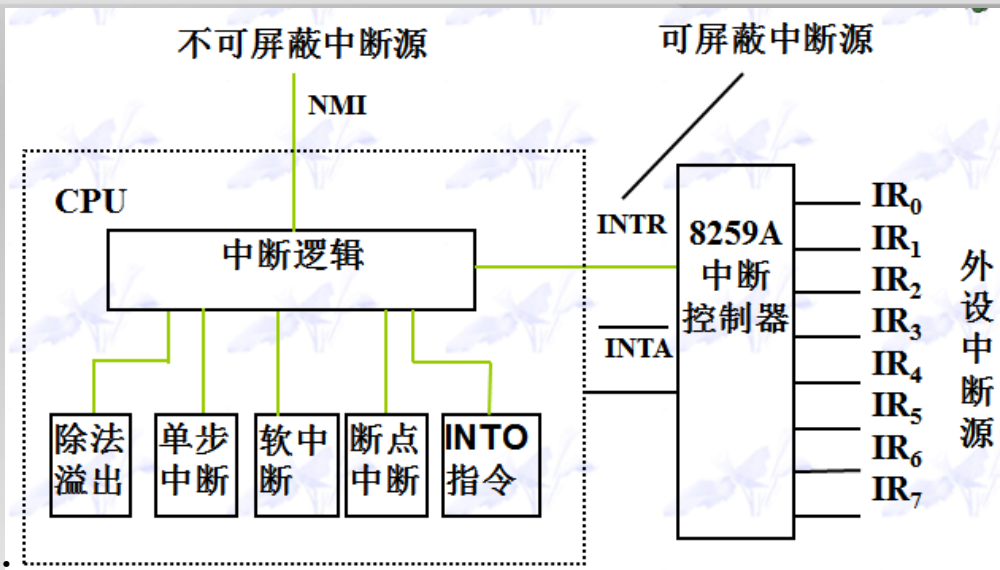
x86启动顺序 – 实模式 (history)

段基地址: 0x1234

段内偏移: 0x4321

	0x	1	2	3	4	0
+	0x		4	3	2	1
<hr/>						
	0x	1	6	6	6	1

物理地址: 0x16661



•取得中断类型码N；

•pushf

•TF=0, IF=0

•push CS

•push IP

•IP=N*4, CS=N*4+2, ; 转中断服务程序

1. 中断请求

2. 中断响应

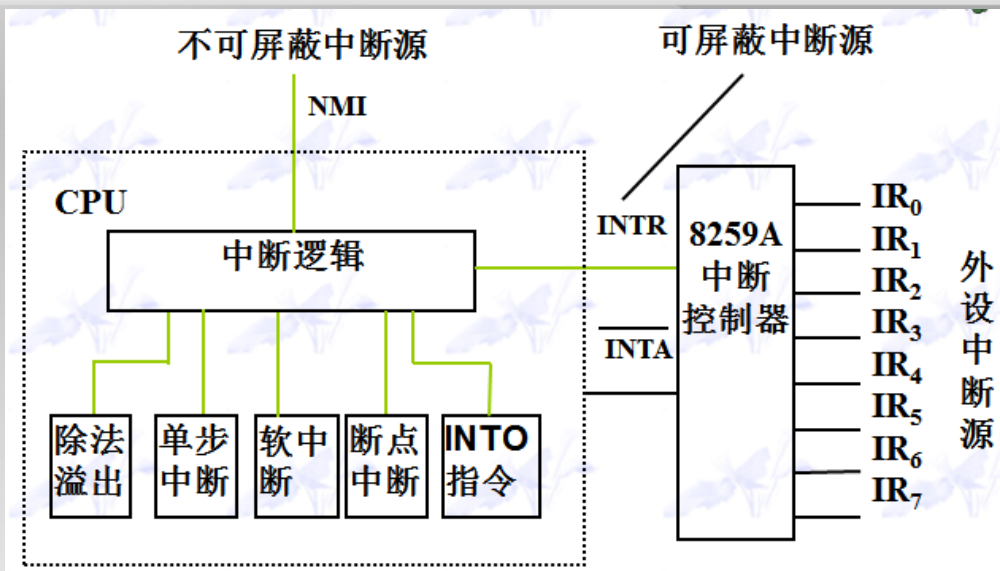
x86启动顺序 – 实模式 (history)

段基地址: 0x1234

段内偏移: 0x4321

	0x	1	2	3	4	0
+	0x		4	3	2	1
<hr/>						
	0x	1	6	6	6	1

物理地址: 0x16661



1. 中断请求
2. 中断响应
3. 保存用到的寄存器
4. 处理中断
5. 恢复用到的寄存器用
6. `iret`指令返回

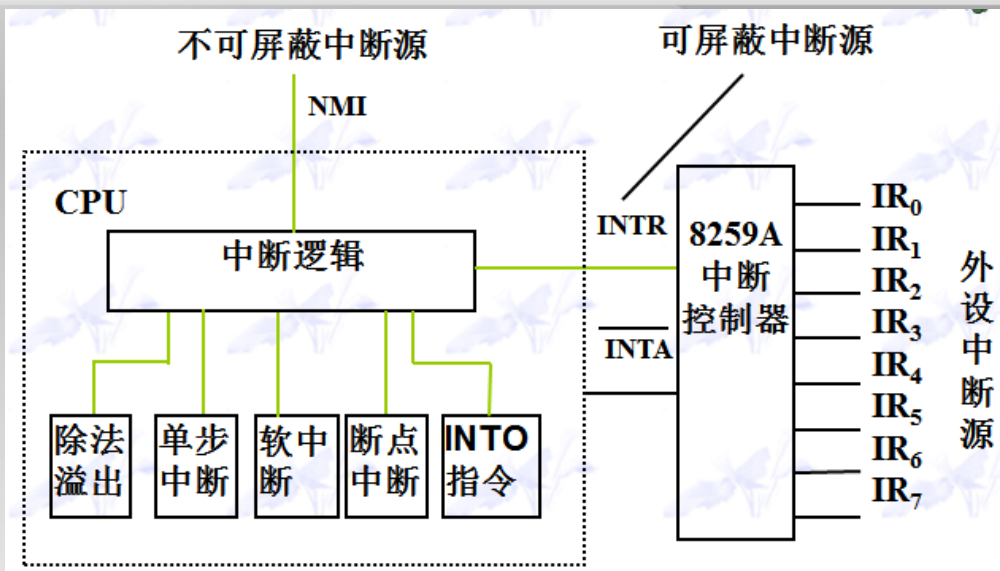
x86启动顺序 – 实模式 (history)

段基地址: 0x1234

段内偏移: 0x4321

	0x	1	2	3	4	0
+	0x		4	3	2	1
<hr/>						
	0x	1	6	6	6	1

物理地址: 0x16661



1. 中断请求
2. 中断响应

1. 保存用到的寄存器
2. 处理中断
3. 恢复用到的寄存器用
4. `iret`指令返回

1. 中断向量表的建立和初始化工作是由BIOS在计算机启动时完成的。
2. 当计算机启动后，操作系统和用户程序再根据自己的需要，修改某些中断的入口地址，使它指向自己的代码。

x86启动顺序 – 实模式 (history)

段基地址:0x1234

段内偏移:0x4321

	0x	1	2	3	4	0
+	0x		4	3	2	1

0x 1 6 6 6 1

物理地址:0x16661

1. 同一个物理地址可以有多种表示方法。例如0x01C0:0x0000和0x0000:0x1C00所表示的物理地址都是0x01C00。

2. 地址空间缺乏保护机制。对于每一个由段寄存器的内容确定的“基地址”，一个进程总是能够访问从此开始64KB的连续地址空间，而无法加以限制。

8086->>80186->>80286->>80386

x86启动顺序 – 段机制

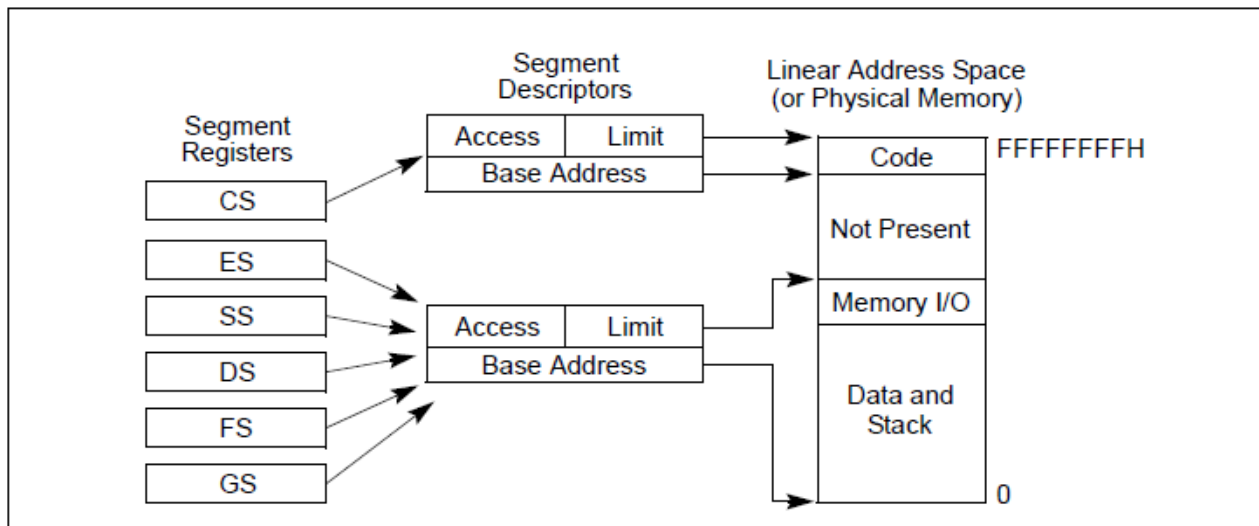


Figure 3-3. Protected Flat Model

摘自"IA-32 Intel体系结构软件开发手册"

x86启动顺序 – 段机制

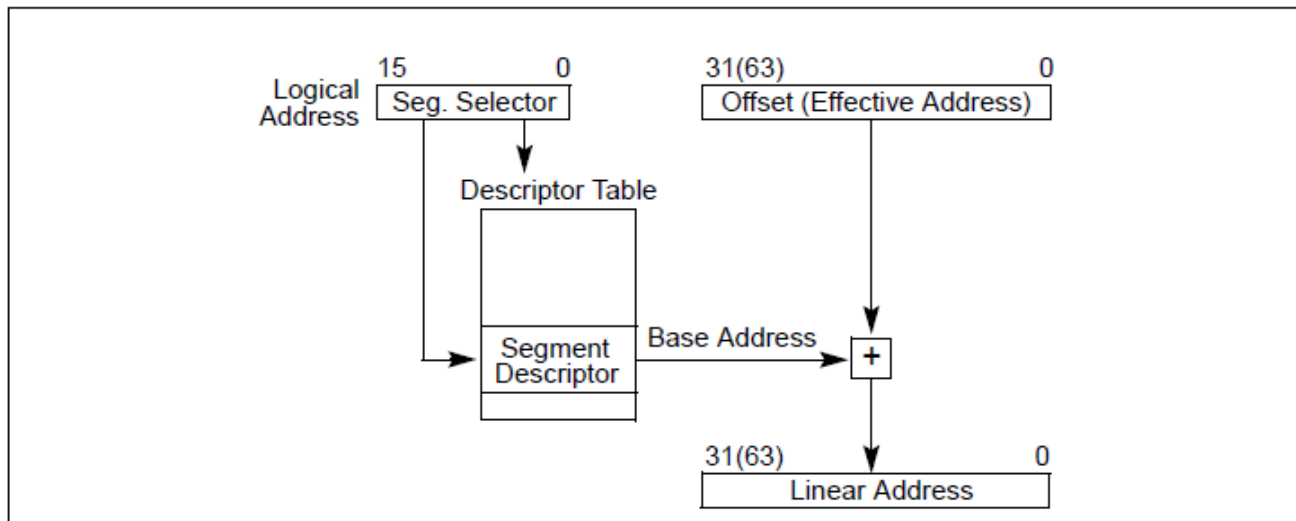
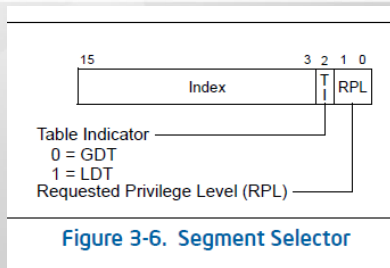


Figure 3-5. Logical Address to Linear Address Translation

摘自"IA-32 Intel体系结构软件开发手册"

x86启动顺序 – 段机制



Loading GDT:

```
lgdt gdt desc
```

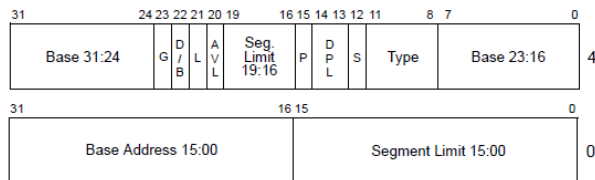
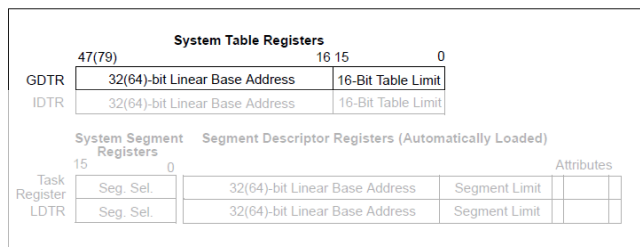
```
gdt:
```

```
.....
```

```
gdt desc:
```

```
.word 0x17
```

```
.long gdt
```



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure 3-8. Segment Descriptor

摘自"IA-32 Intel体系结构软件开发手册"

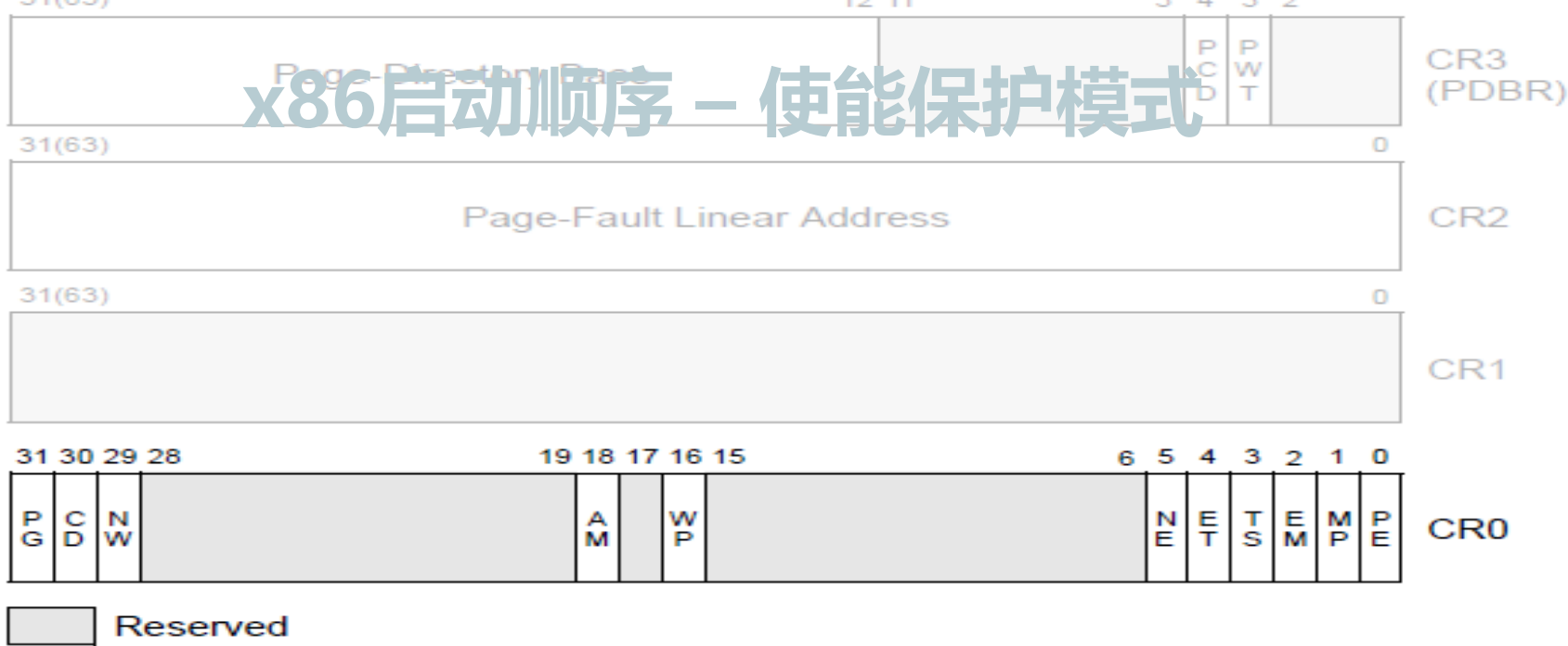


Figure 2-7. Control Registers

摘自"IA-32 Intel体系结构软件开发手册"

- 使能保护模式 (protection mode) , bootloader/OS 要设置 CR0的bit 0 (PE)
- 段机制 (Segment-level protection) 在保护模式下是自动使能的

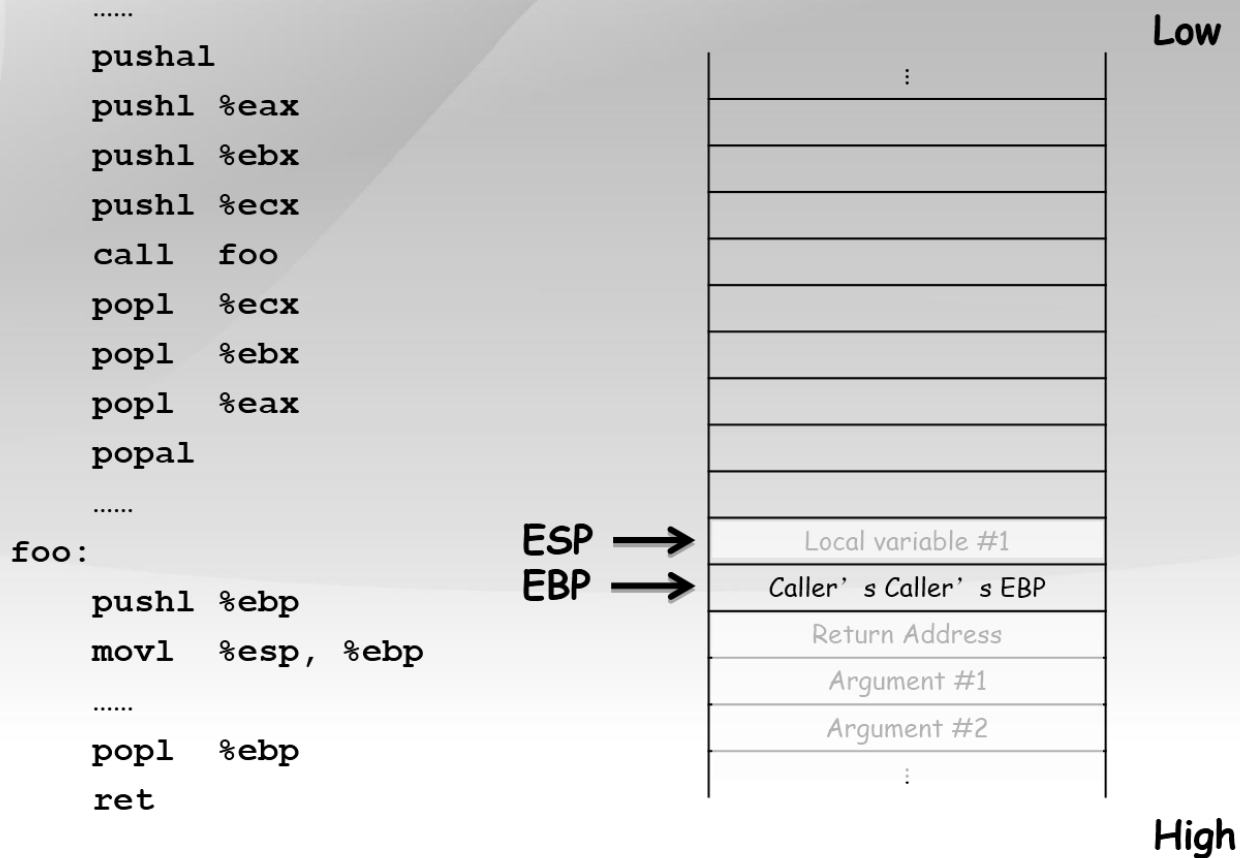
x86启动顺序 – 参考资料

- Chap. 2.5 (Control Registers)), Vol. 3, Intel® and IA-32 Architectures Software Developer' s Manual
- Chap. 3 (Protected-Mode Memory Management), Vol. 3, Intel® and IA-32 Architectures Software Developer' s Manual
- Chap. 9.1 (Initialization Overview), Vol. 3, Intel® and IA-32 Architectures Software Developer' s Manual
- An introduction to ELF format: <http://wiki.osdev.org/ELF>

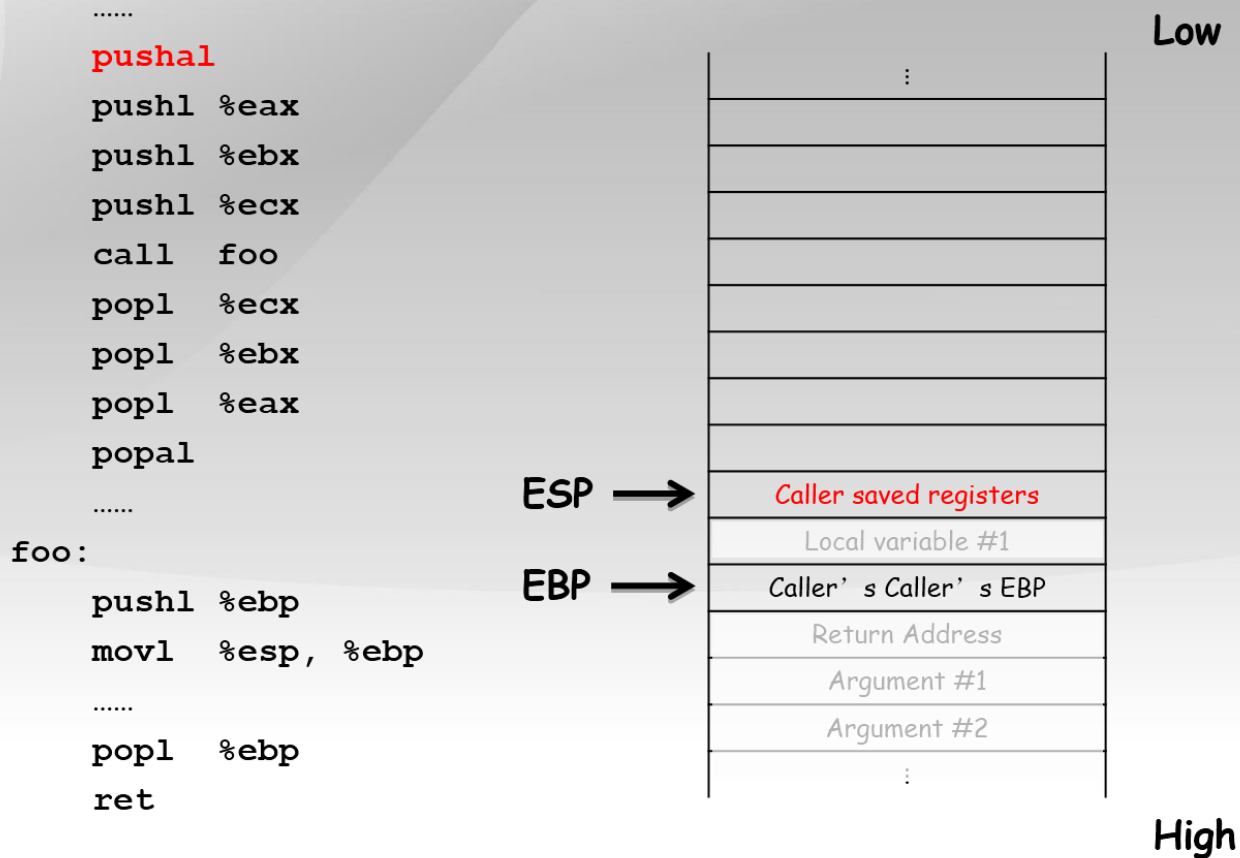
- 理解C函数调用在汇编级是如何实现的
- 理解如何在汇编级代码中调用C函数
- 理解基于EBP寄存器的函数调用栈

C函数调用的实现

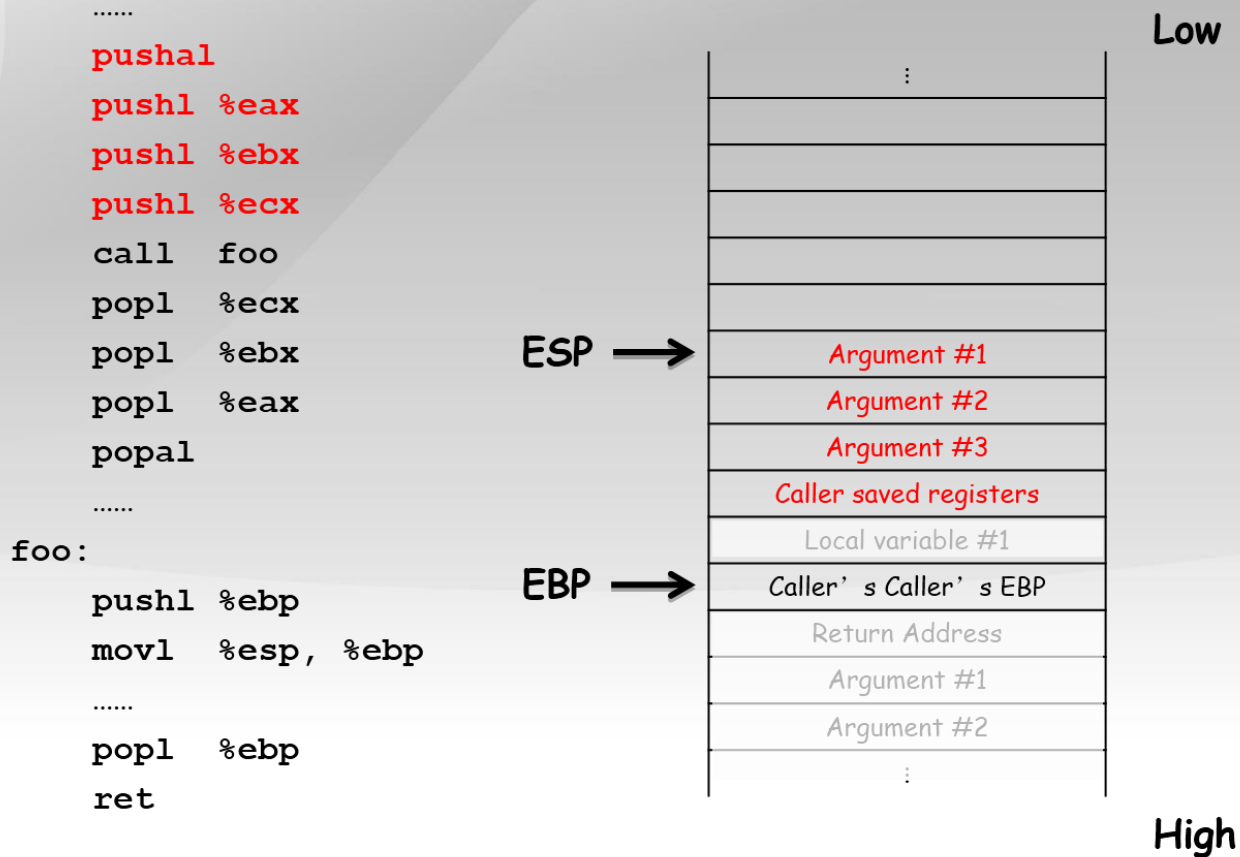
C函数调用的实现



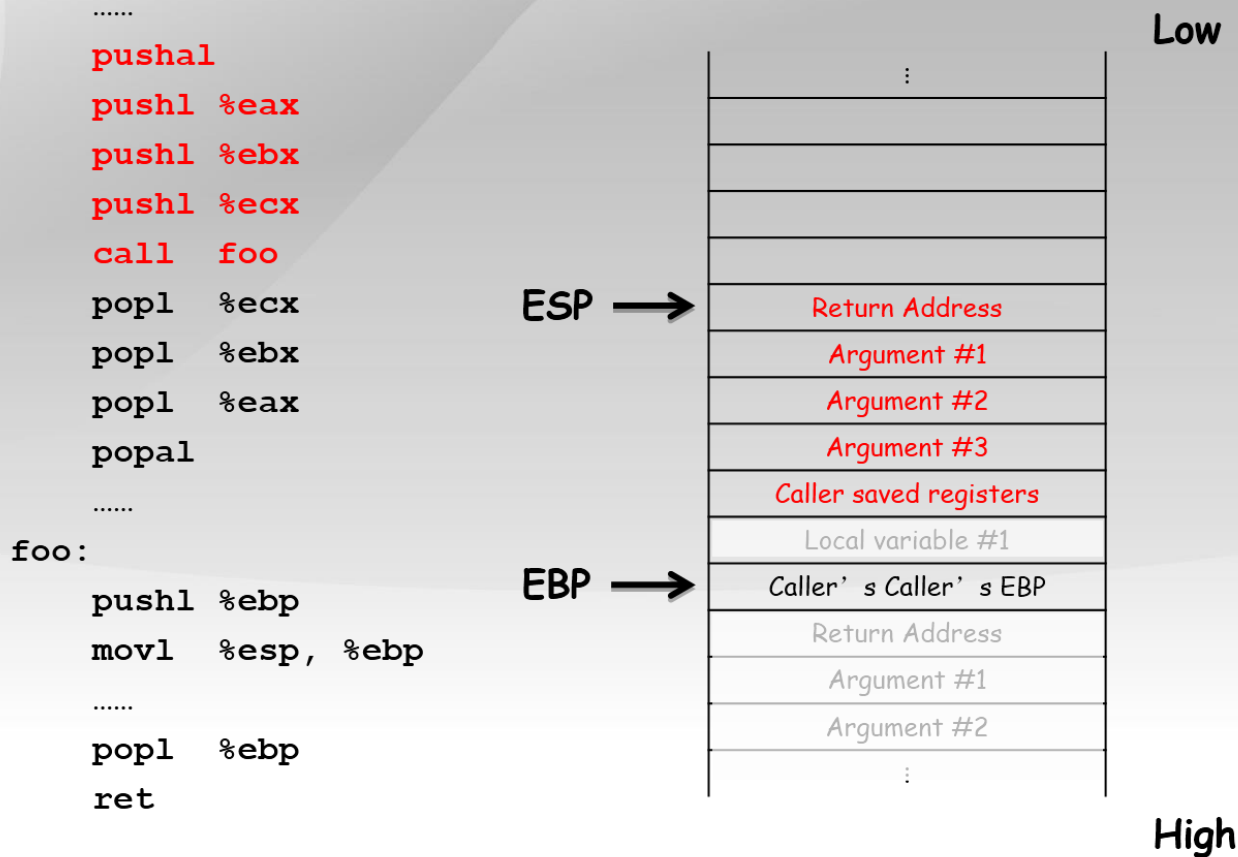
C函数调用的实现



C函数调用的实现

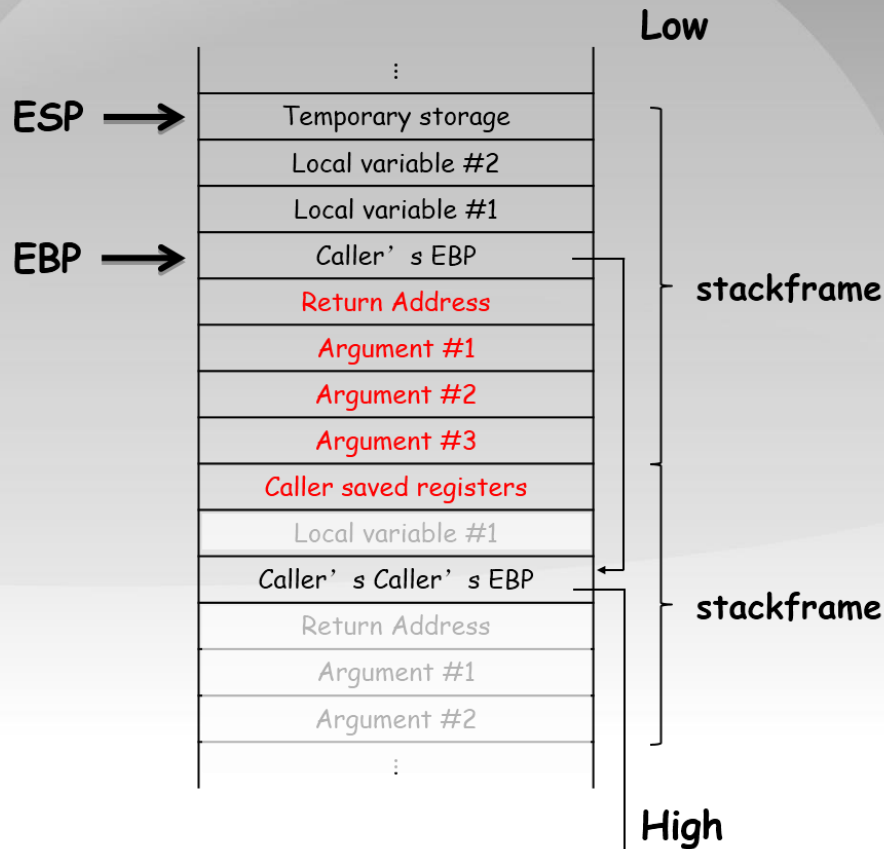


C函数调用的实现

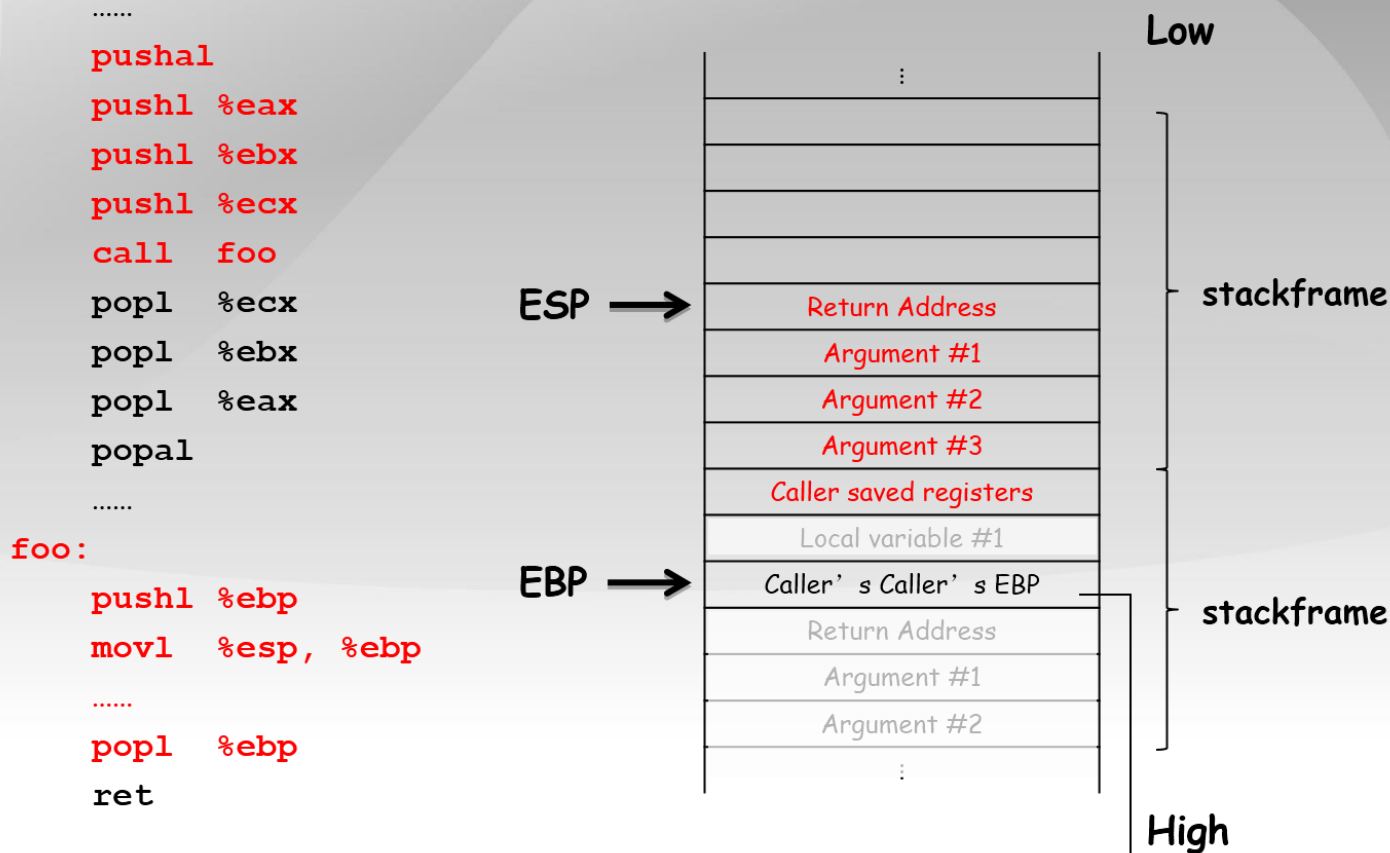


C函数调用的实现

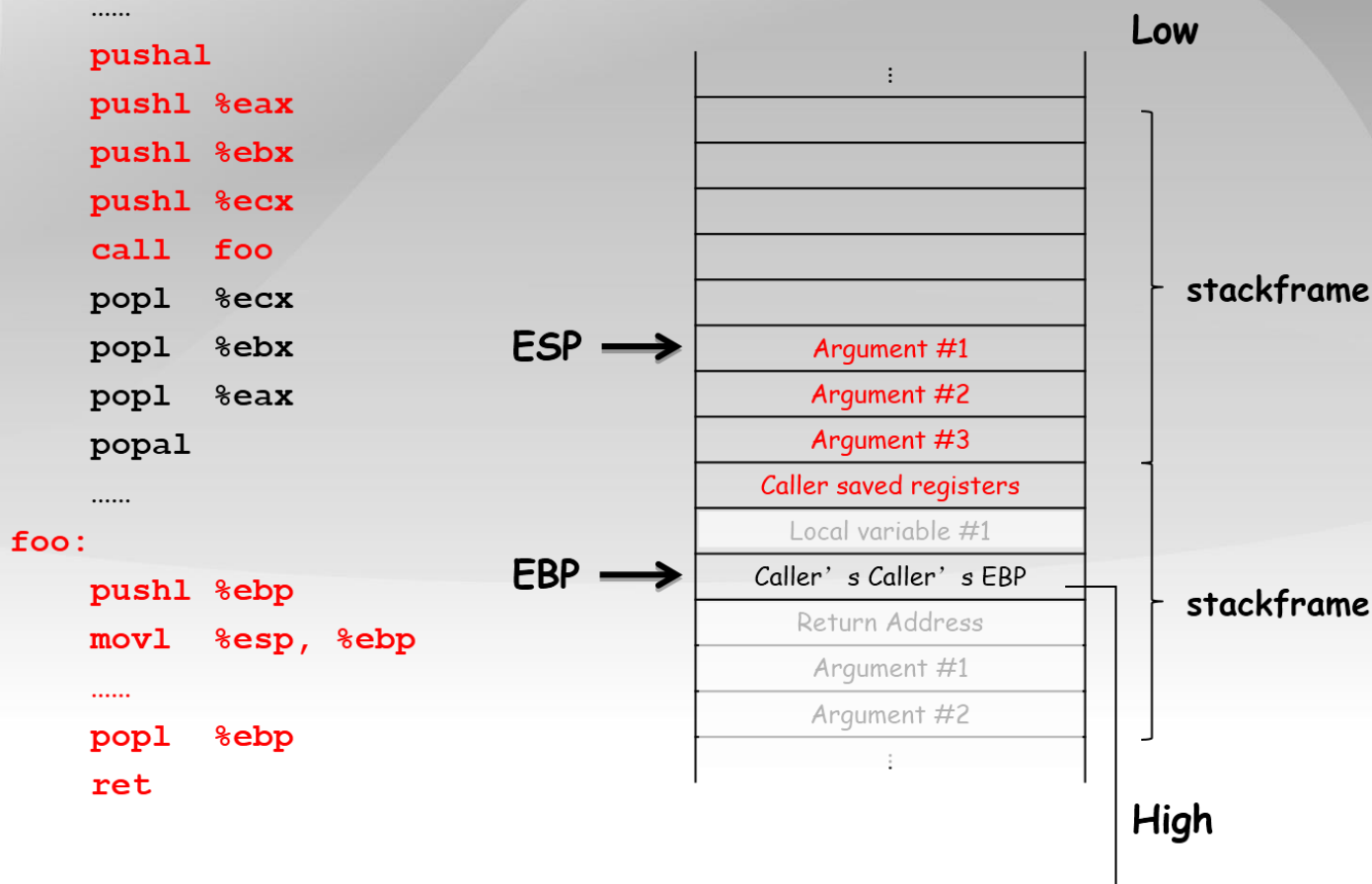
```
.....  
pushal  
pushl %eax  
pushl %ebx  
pushl %ecx  
call  foo  
popl  %ecx  
popl  %ebx  
popl  %eax  
popal  
.....  
foo:  
pushl %ebp  
movl  %esp, %ebp  
.....  
popl  %ebp  
ret
```



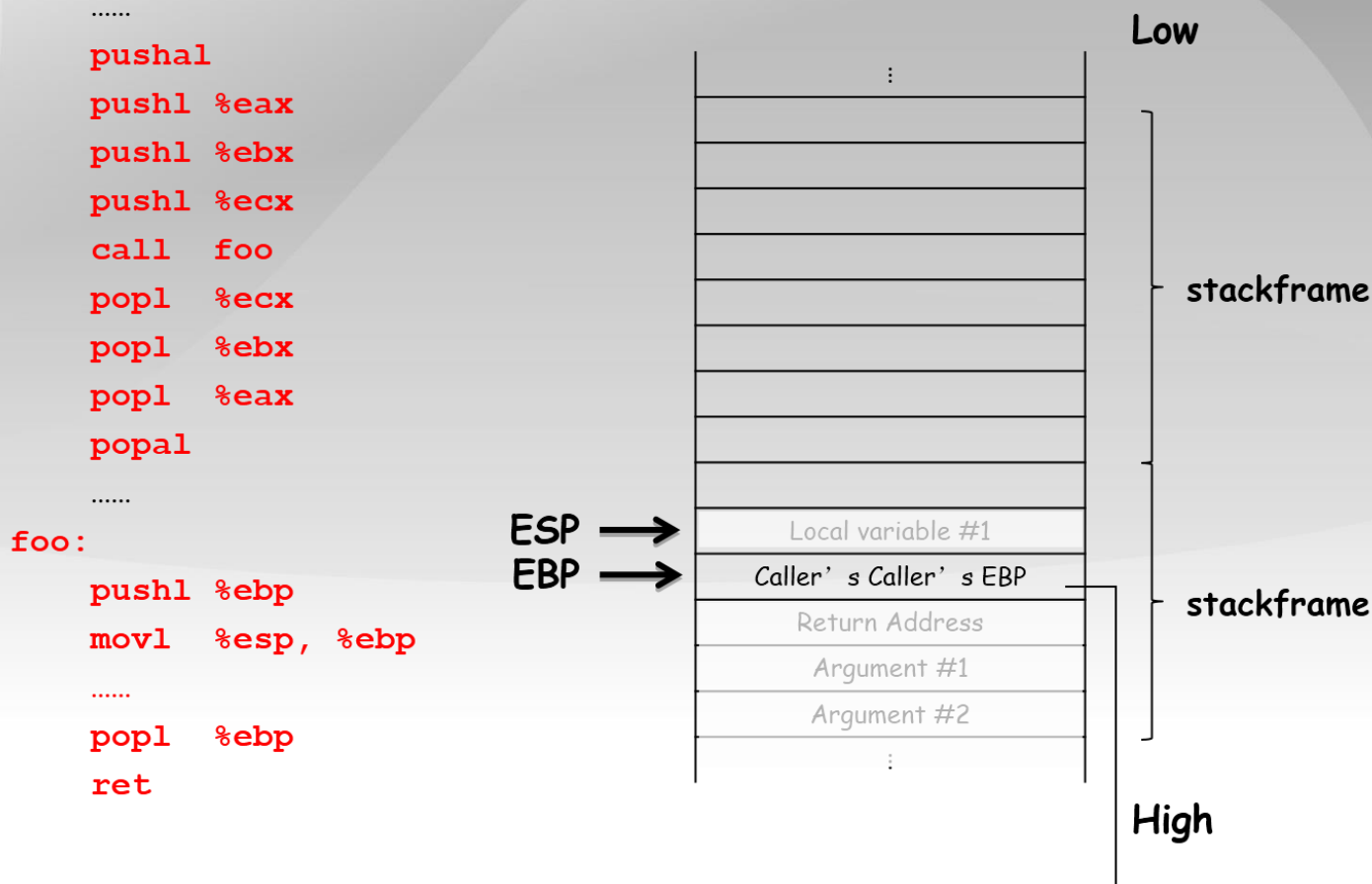
C函数调用的实现



C函数调用的实现



C函数调用的实现



C函数调用的实现

■ 其他需要注意的事项

- 参数 (parameters) & 函数返回值 (return values) 可通过寄存器或位于内存中的栈来传递
- 不需要保存/恢复(save/restore)所有寄存器

C函数调用的——参考资料

- Understanding the Stack:

<http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/stack.html>

▣ 阅读理解内联汇编 (inline assembly instructions)

GCC内联汇编 INLINE ASSEMBLY

CC内联汇编

- 什么是内联汇编 (Inline assembly) ?
 - ▣ 这是GCC对C语言的扩展
 - ▣ 可直接在C语句中插入汇编指令
- 有何用处?
 - ▣ 调用C语言不支持的指令
 - ▣ 用汇编在C语言中手动优化
- 如何工作?
 - ▣ 用给定的模板和约束来生成汇编指令
 - ▣ 在C函数内形成汇编源码

CC内联汇编– Example 1

Assembly (*.S):

```
movl $0xffff, %eax
```



Inline assembly (*.c):

```
asm ( "movl $0xffff, %%eax\n" )
```

CC内联汇编– Syntax

asm (assembler template

: output operands (optional)

: input operands (optional)

: clobbers (optional)

);

CC内联汇编– Example 2

Inline assembly (*.c):

```
uint32_t cr0;  
asm volatile ("movl %%cr0, %0\n" : "=r" (cr0));  
cr0 |= 0x80000000;  
asm volatile ("movl %0, %%cr0\n" :: "r" (cr0));
```



Generated assembly code (*.s):

```
movl %cr0, %ebx  
movl %ebx, 12(%esp)  
orl $-2147483648, 12(%esp)  
movl 12(%esp), %eax  
movl %eax, %cr0
```

CC内联汇编– Example 2

Inline assembly (*.c):

```
uint32_t cr0;  
asm volatile ("movl %%cr0, %0\n" : "=r" (cr0)) ;  
cr0 |= 0x80000000;  
asm volatile ("movl %0, %%cr0\n" :: "r" (cr0)) ;
```

- **volatile**

No reordering; No elimination

- **%0**

The first constraint following

- **r**

A constraint; GCC is free to use any register

CC内联汇编– Example 3

```
long __res, arg1 = 2, arg2 = 22, arg3 = 222, arg4 = 233;  
__asm__ volatile ("int $0x80"  
    : "=a" (__res)  
    : "0" (11), "b" (arg1), "c" (arg2), "d" (arg3), "S" (arg4)) ;
```



```
.....  
movl    $11, %eax  
movl    -28(%ebp), %ebx  
movl    -24(%ebp), %ecx  
movl    -20(%ebp), %edx  
movl    -16(%ebp), %esi  
int     $0x80  
movl    %eax, %edi  
movl    %edi, -12(%ebp)
```

- Constraints

a = %eax

b = %ebx

c = %ecx

d = %edx

S = %esi

D = %edi

0 = same as the first

CC内联汇编- 参考资料

- GCC Manual 6.41 – 6.43
- Inline assembly for x86 in Linux:

<http://www.ibm.com/developerworks/library/l-ia/index.html>

- 了解x86中的中断源
- 了解CPU与操作系统如何处理中断
- 能够对中断向量表（中断描述符表，简称IDT）进行初始化

X86中的中断处理

X86中的中断处理- 中断源

■ 中断 Interrupts

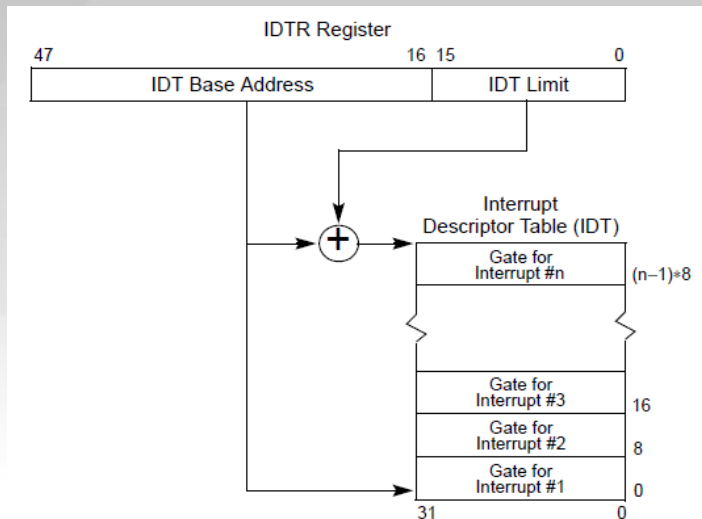
- ▶ 外部中断 External (hardware generated) interrupts
串口、硬盘、网卡、时钟、...
- ▶ 软件产生的中断 Software generated interrupts
The INT n 指令，通常用于系统调用

异常 Exceptions

- ▶ 程序错误
- ▶ 软件产生的异常 Software generated exceptions
INTO, INT 3 and BOUND
- ▶ 机器检查出的异常S

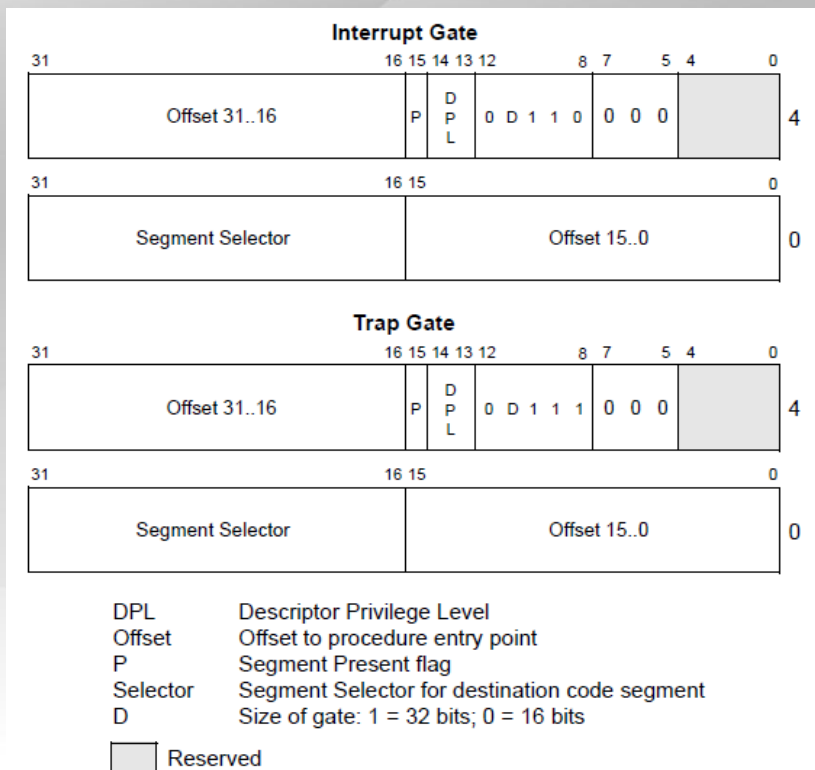
X86中的中断处理- 确定中断服务例程 (ISR)

- 每个中断或异常与一个中断服务例程 (Interrupt Service Routine , 简称ISR) 关联 , 其关联关系存储在中断描述符表 (Interrupt Descriptor Table , 简称IDT) 。
- IDT的起始地址和大小保存在中断描述符表寄存器IDTR中



摘自"IA-32 Intel体系结构软件开发手册"

X86中的中断处理- 确定中断服务例程 (ISR)



摘自"IA-32 Intel体系结构软件开发手册"

X86中的中断处理- 确定中断服务例程 (ISR)

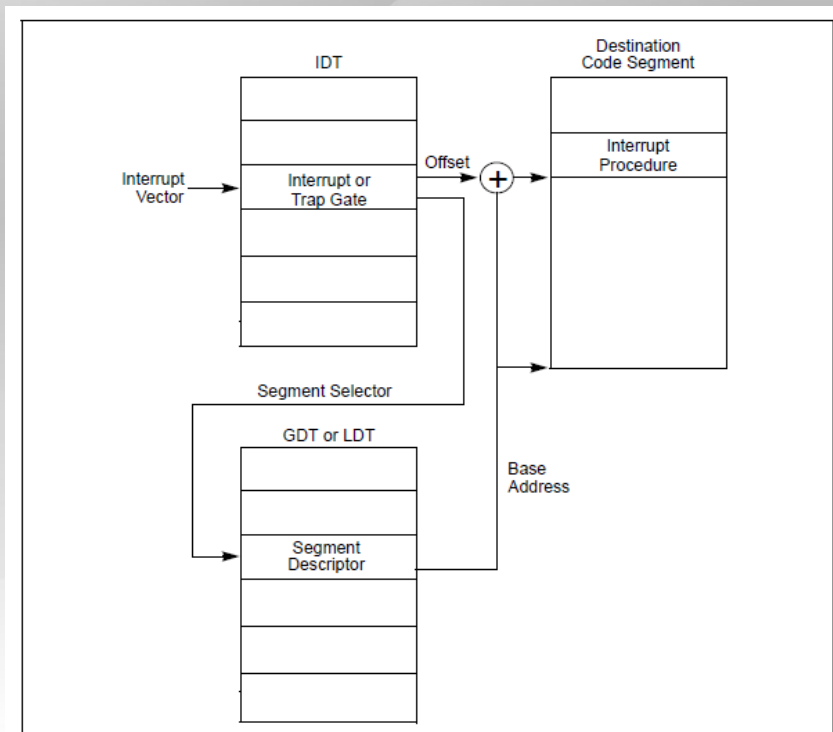
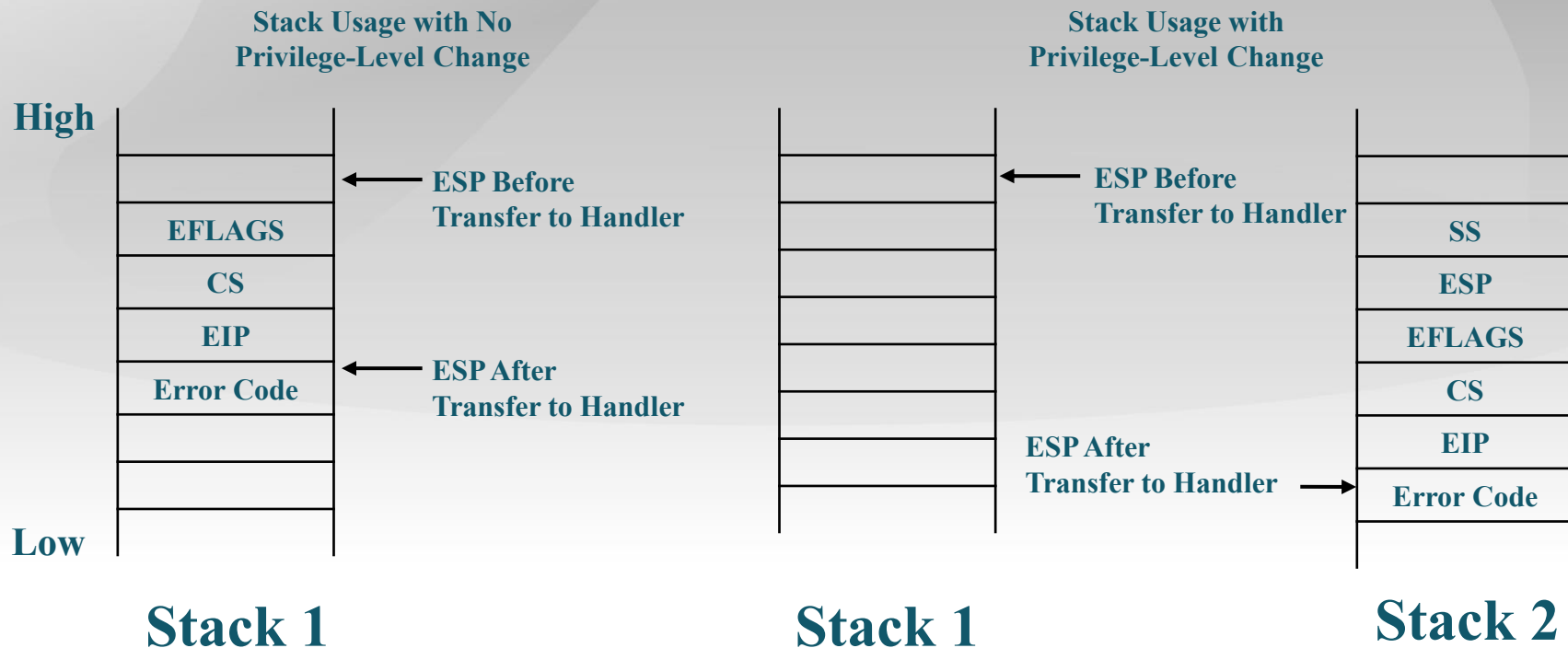


Figure 6-3. Interrupt Procedure Call

摘自"IA-32 Intel体系结构软件开发手册"

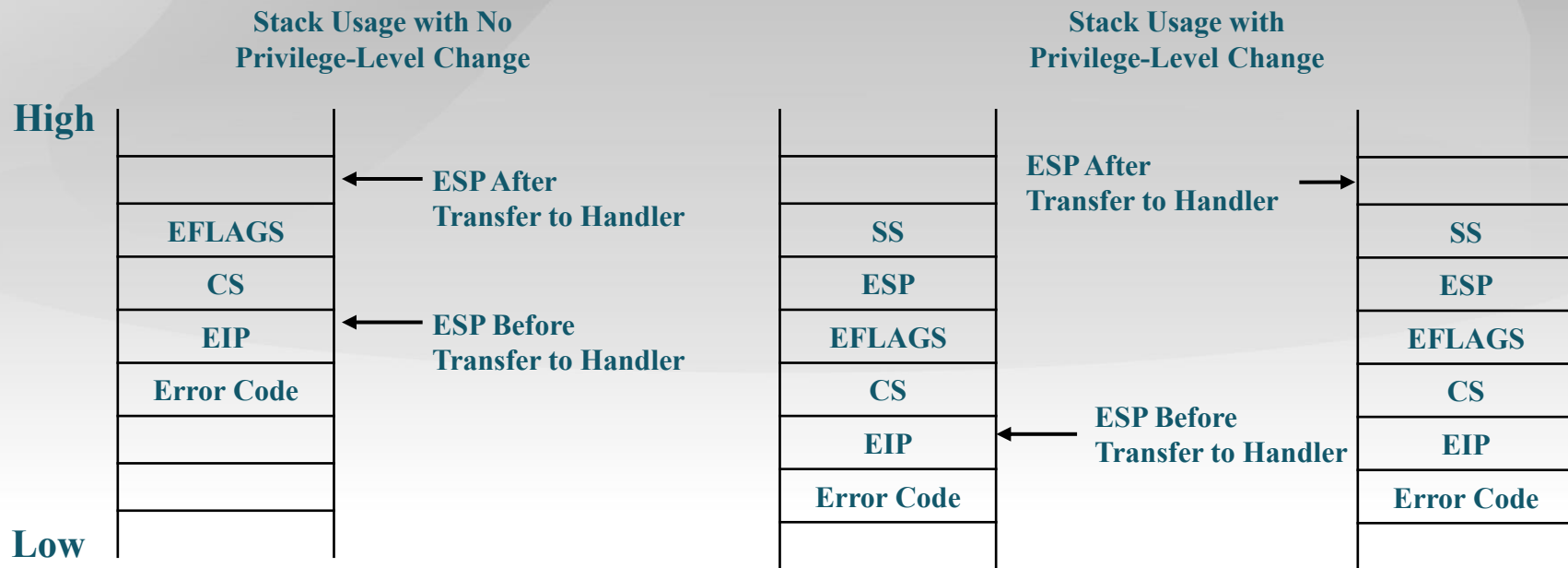
X86中的中断处理—切换到中断服务例程（ISR）

■ 不同特权级的中断切换对堆栈的影响



X86中的中断处理-从中断服务例程（ISR）返回

- **iret vs. ret vs. retf** : iret 弹出 EFLAGS 和 SS/ESP（根据是否改变特权级），但 ret弹出EIP，retf弹出CS和EIP



X86中的中断处理-系统调用

- 用户程序通过系统调用访问OS内核服务。
- 如何实现
 - ▶ 需要指定中断号
 - ▶ 使用Trap , 也称Software generated interrupt
 - ▶ 或使用特殊指令 (SYSENTER/SYSEXIT)

X86中的中断处理-参考资料

- Chap. 6, Vol. 3, Intel® and IA-32 Architectures Software Developer's Manual