Chp3 机器级程序设计

第2讲

主要内容

- 数据传送
- 算术和逻辑操作
- 条件、循环、分支控制
- **堆栈、过程、递归和指针**
- **数组、结构、联合**
- 内存分配、缓冲区溢出

过程控制

- 使用栈来支持过程调用和返回
- 过程调用:`call label`
  - **返回地址进栈**; 跳转到*label处*
- 返回地址值：Call后面一条指令的地址
  - 反汇编例子

```
804854e: e8 3d 06 00 00      call   8048b90 <main>
8048553: 50                  pushl  %eax
```

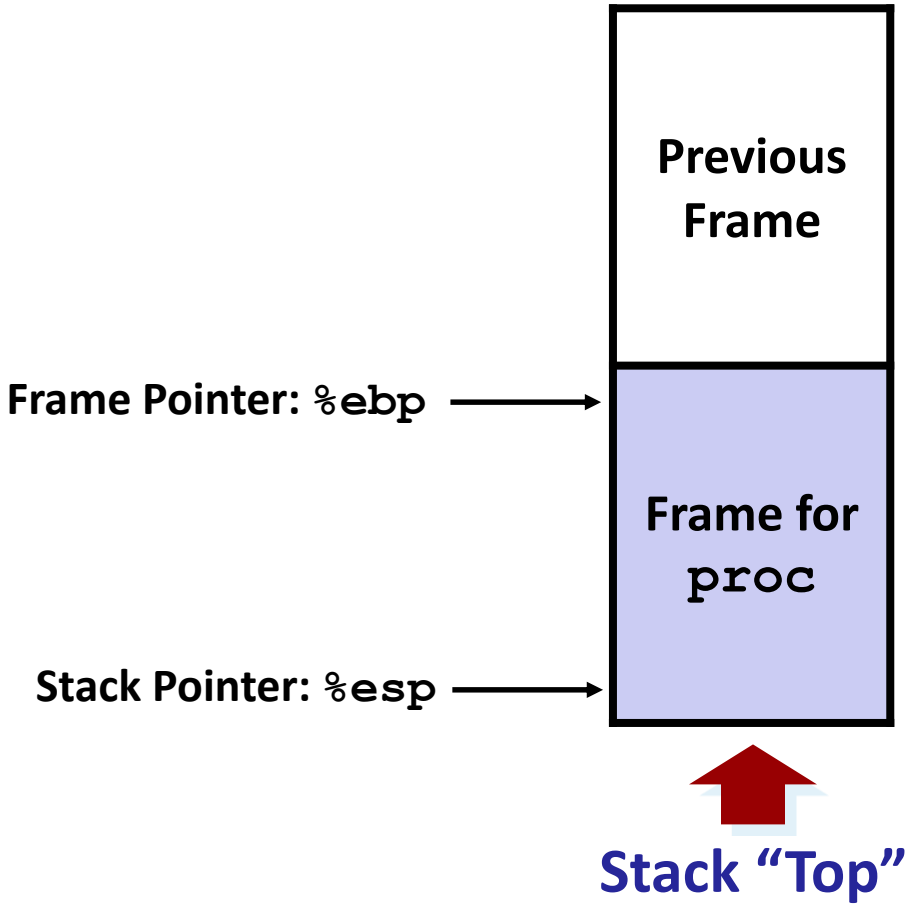
  - 返回地址= 0x8048553
- 过程返回:`ret`
  - 从栈中弹出地址; 跳转到地址处

基于栈的语言

- 支持递归的语言
  - 例如：C, Pascal, Java
  - 代码必须是“*可重入*”的
    - 同时存在单个过程的多个运行实例
  - 需要存储每个实例状态的空间
    - 参数、局部变量、返回指针
- 栈规则
  - 给定过程的状态只在有限的时间内需要
    - 从被调用开始，到返回时结束
  - 被调用者先于调用者返回
- 栈按照帧（Frame）进行分配
  - 单一过程实例的状态

栈帧

- 内容
  - 局部变量
  - 返回信息
  - 临时空间
- 管理
  - 在进入过程时分配空间
    - Set-up代码
  - 在过程返回时释放空间
    - Finish代码
- 指针
  - 栈指针%esp 指示栈顶
  - 帧指针 %ebp指示当前帧开始



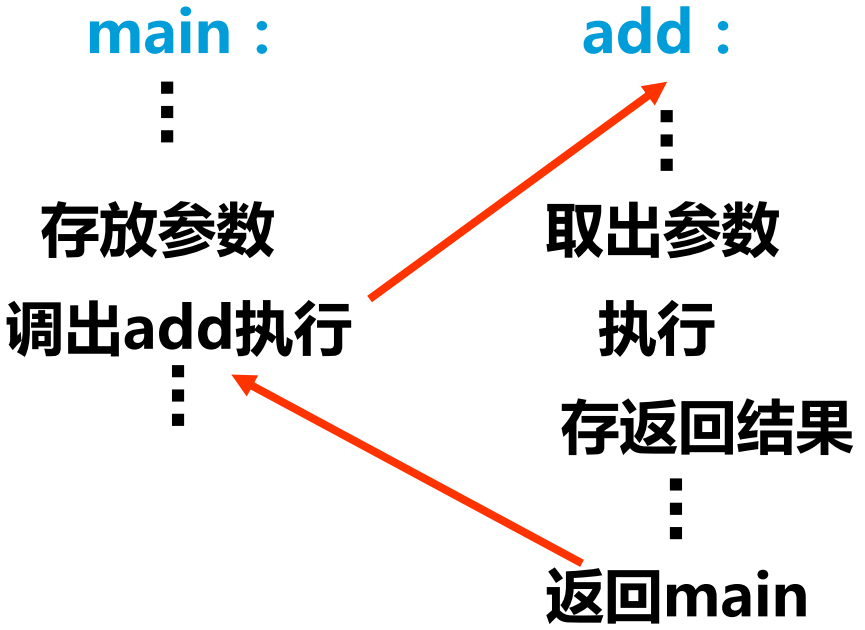
过程调用的机器级表示

- 以下过程（函数）调用对应的机器级代码是什么？
- 如何将t1(125)、t2(80)分别传递给add中的形式参数x、y
- add函数执行的结果如何返回给caller?

```
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

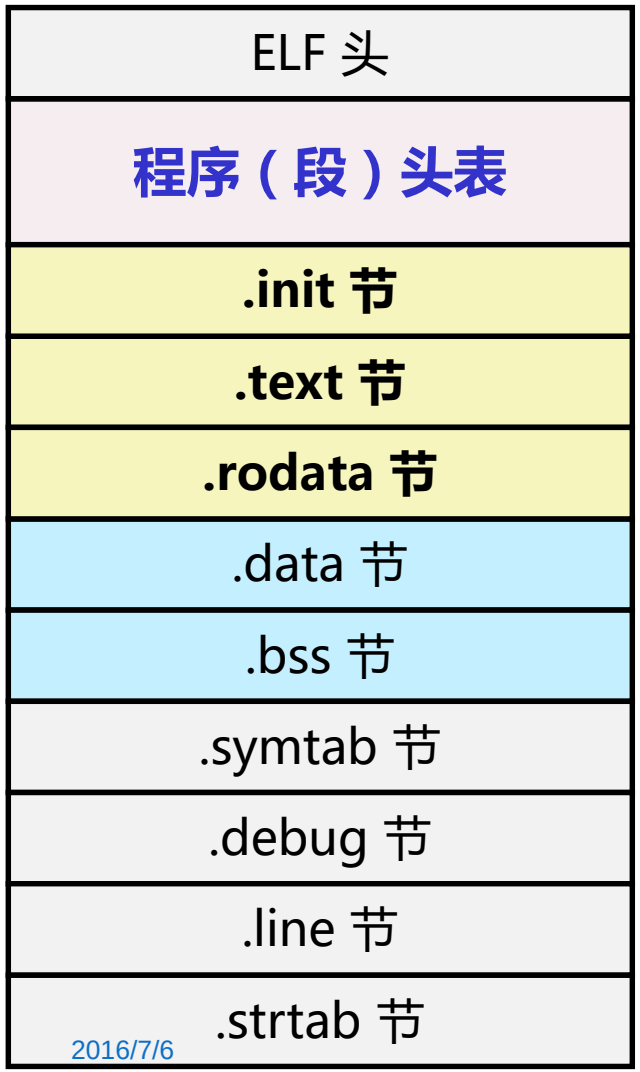


IA-32中参数通过栈（stack）来传

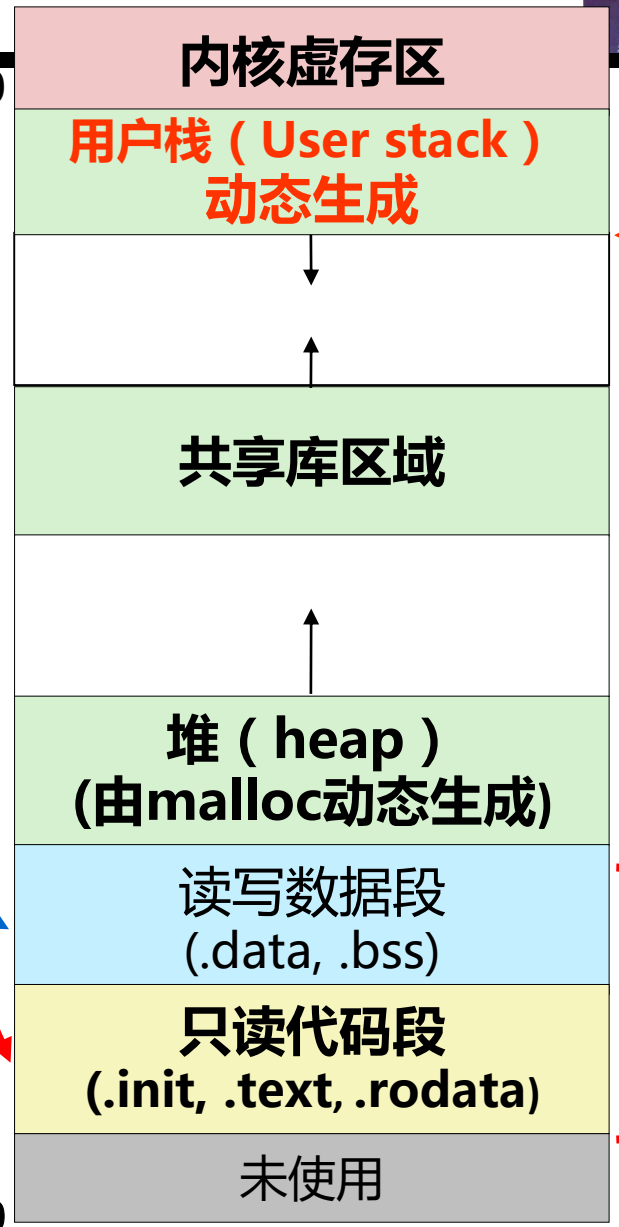
栈（stack）在哪里？

可执行文件的存储器映像

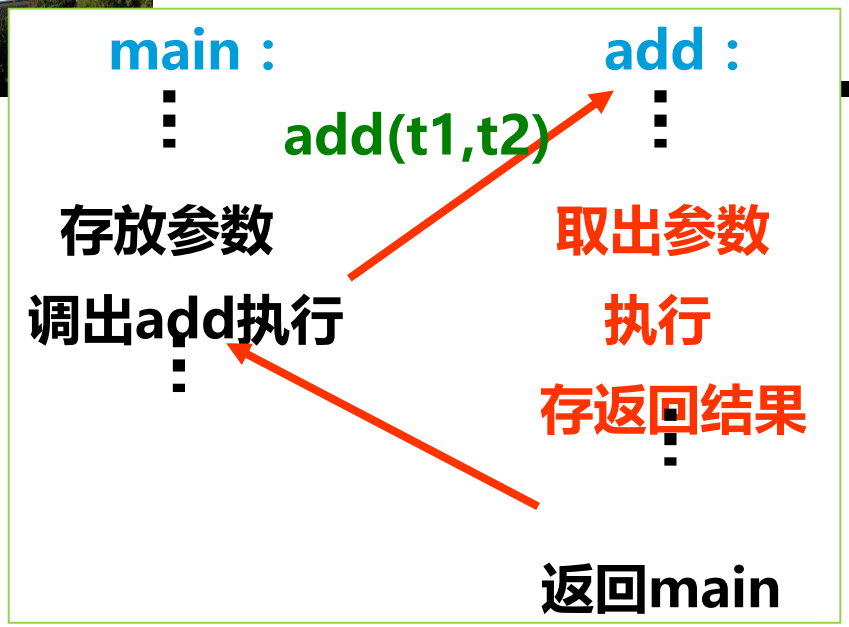
程序(段)头表描述如何映射



从高地址向低地址增长！



过程调用的机器级表示



何为现场？

通用寄存器的内容！

为何要保存现场？

因为所有过程共享一套通用寄存器

过程调用的执行步骤(P为调用者，Q为被调用者)

（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，并为自己的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

过程调用的机器级表示

- IA-32的寄存器使用约定
  - 调用者保存寄存器：EAX、EDX、ECX

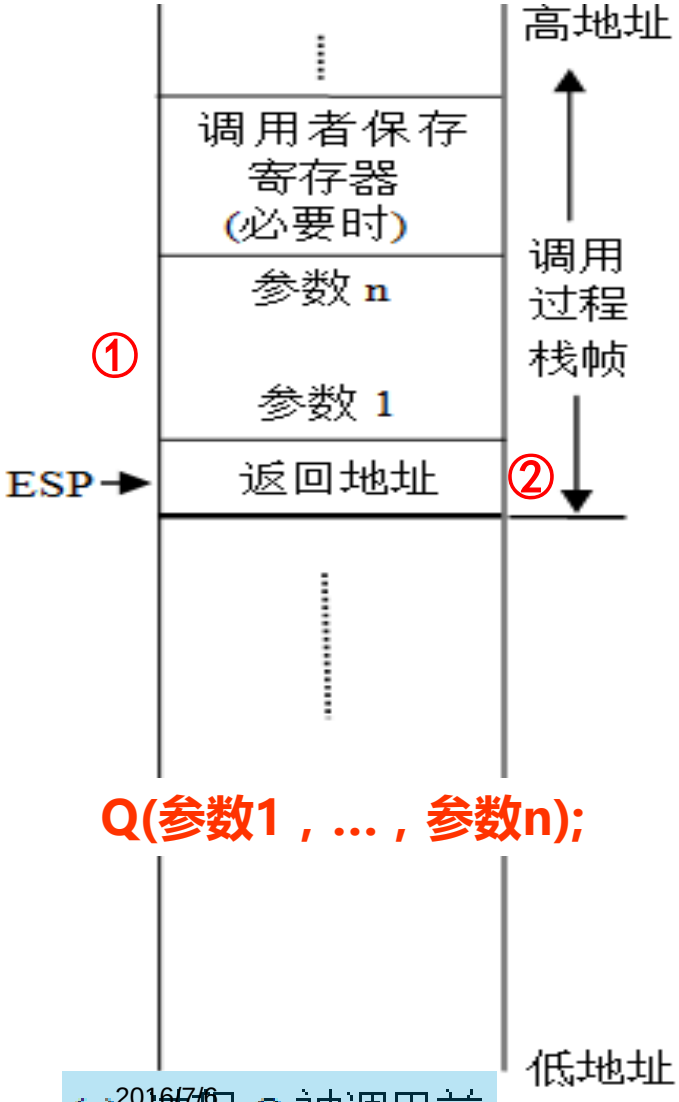
    当过程P调用过程Q时，Q可以直接使用这三个寄存器，不用将它们的值保存到栈中。如果P在从Q返回后还要用这三个寄存器的话，P应在转到Q之前先保存，并在从Q返回后先恢复它们的值再使用。
  - 被调用者保存寄存器：EBX、ESI、EDI

    Q必须先将它们的值保存到栈中再使用它们，并在返回P之前恢复它们的值。
  - EBP和ESP分别是帧指针寄存器和栈指针寄存器，分别用来指向当前栈帧的底部和顶部。

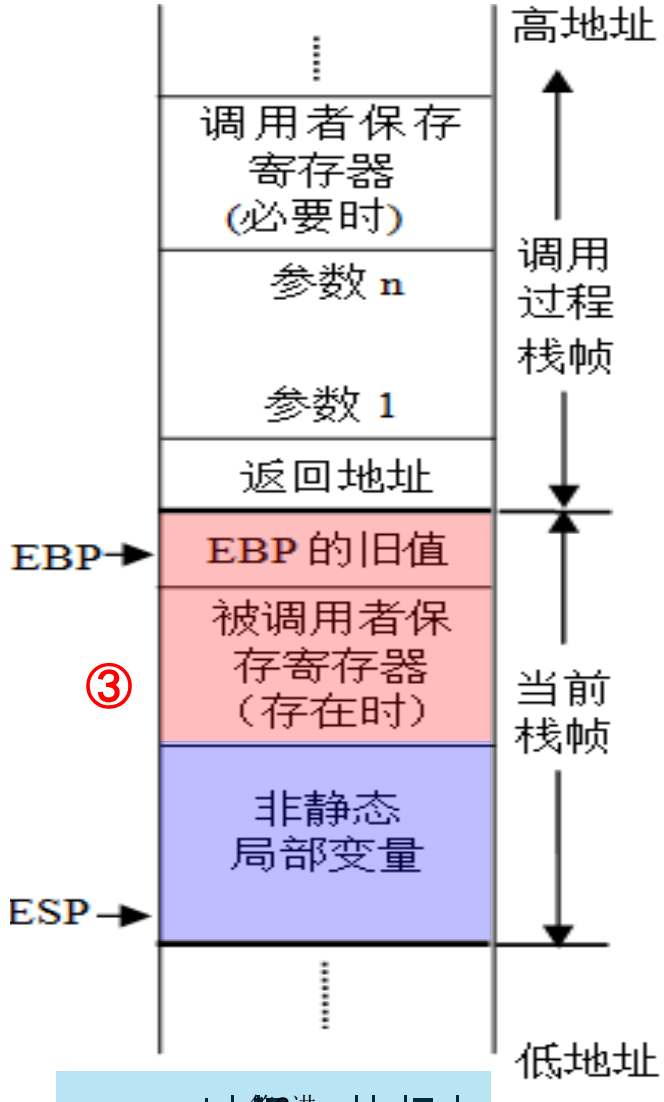
问题：为减少准备和结束阶段的开销，每个过程应先使用哪些寄存器？

EAX、ECX、EDX！

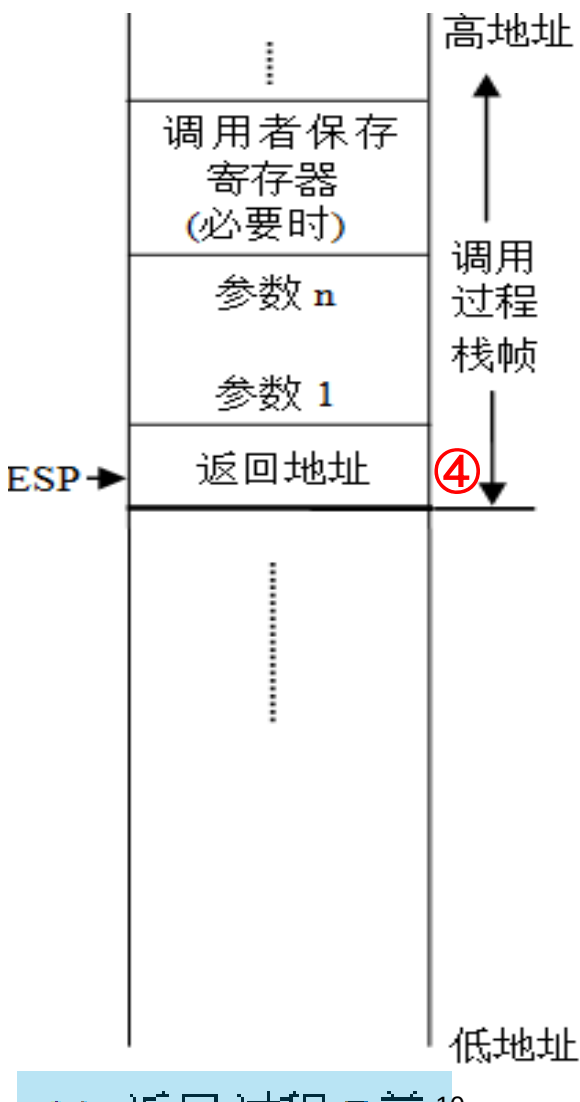
过程调用过程中栈和栈帧的变化



(a) 过程 Q 被调用前



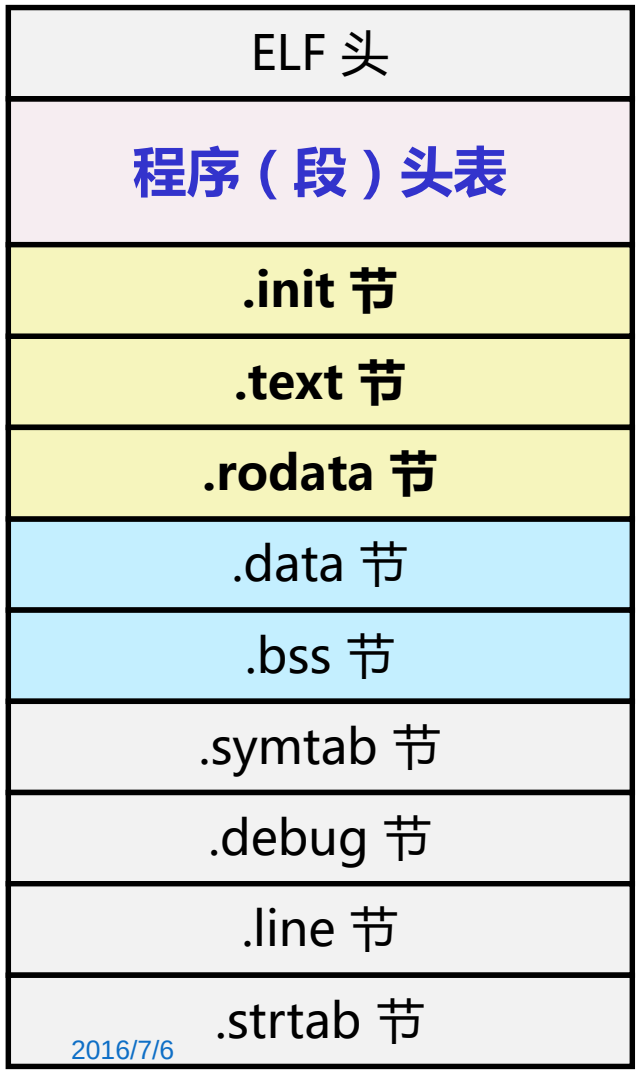
(b) 过程 Q 执行中



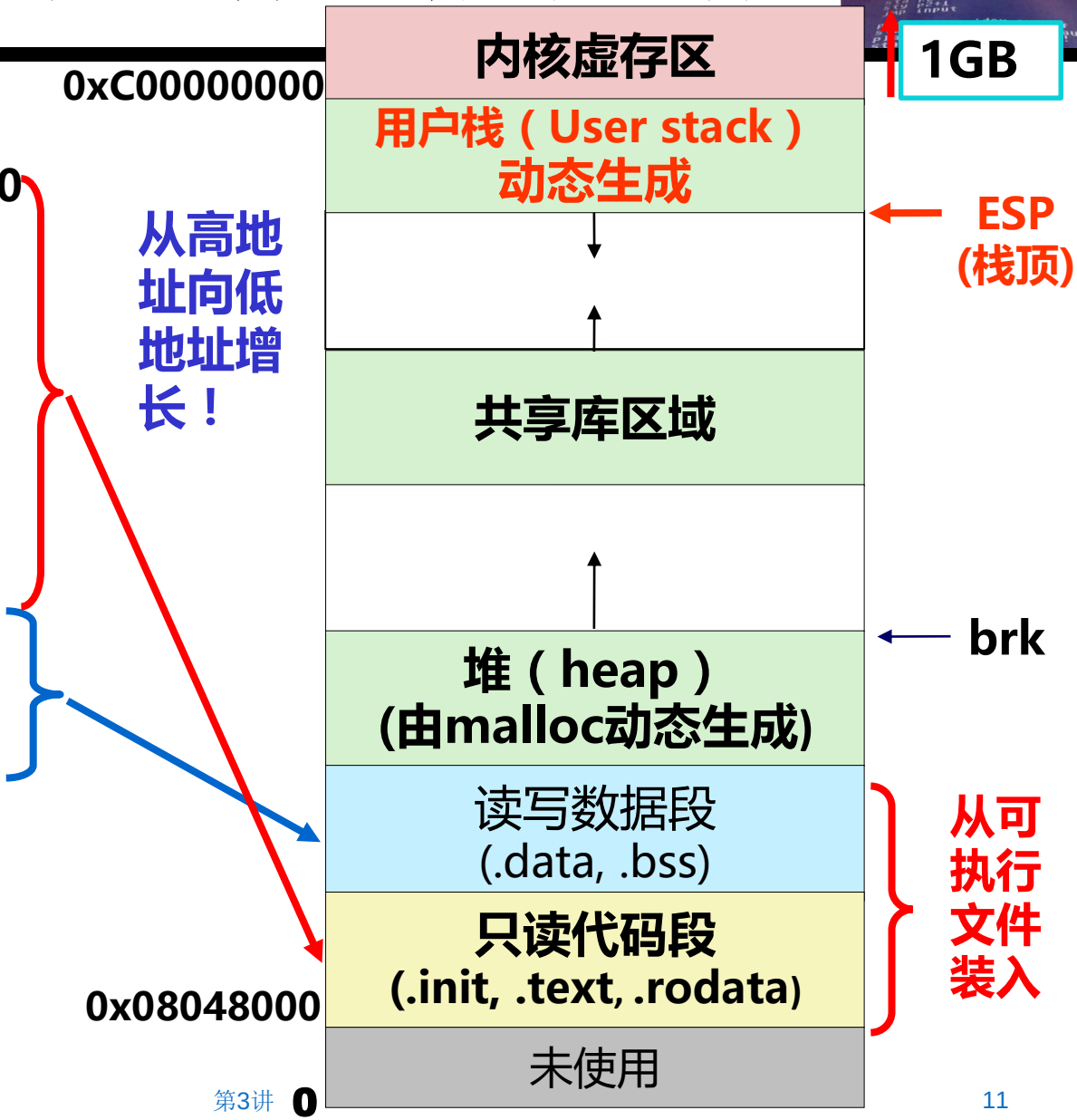
(c) 返回过程 P 前

Linux可执行文件的存储映像

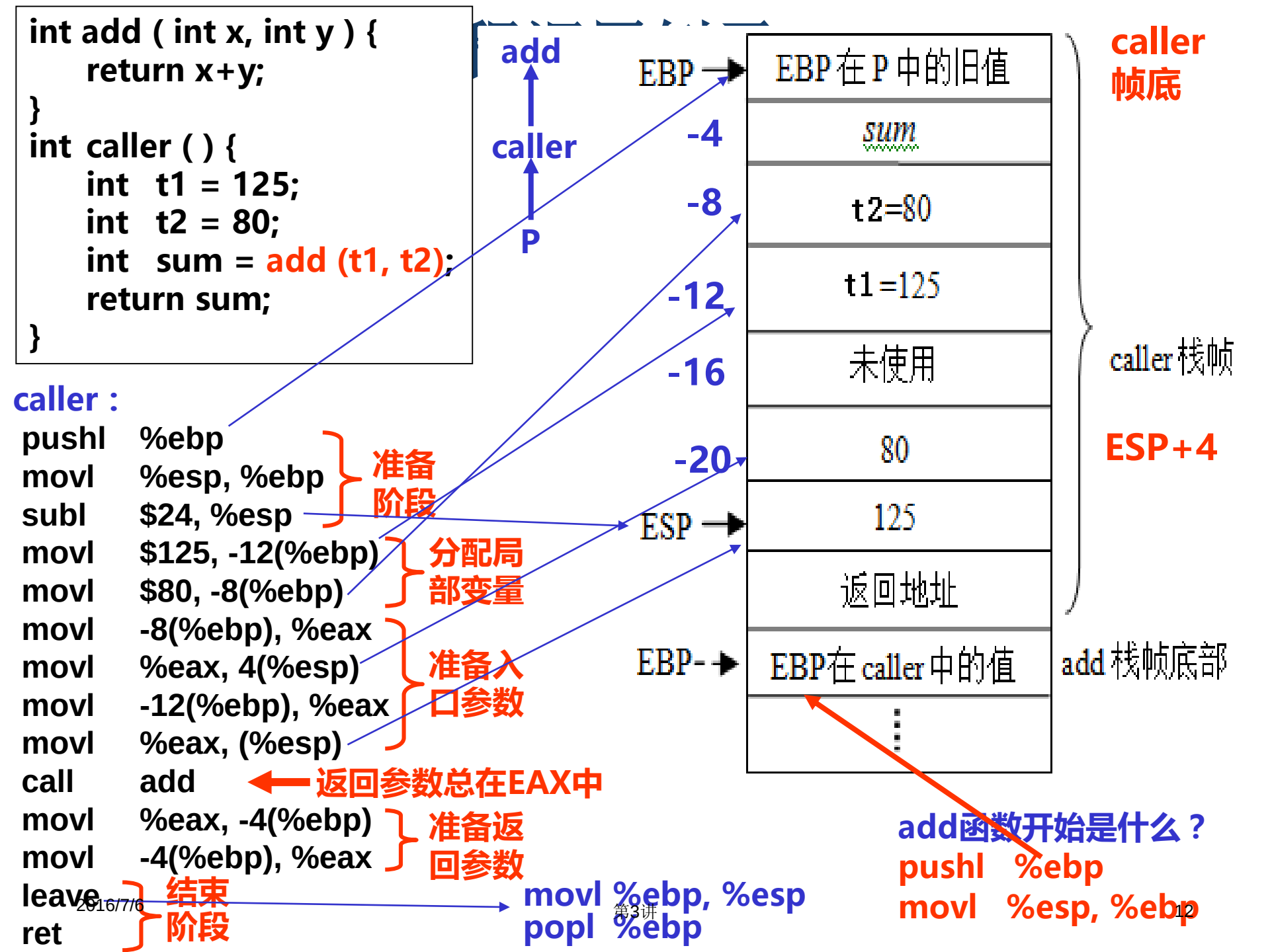
程序(段)头表描述如何映射



从高地址向低地址增长！



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



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

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

add函数开始是什么？

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

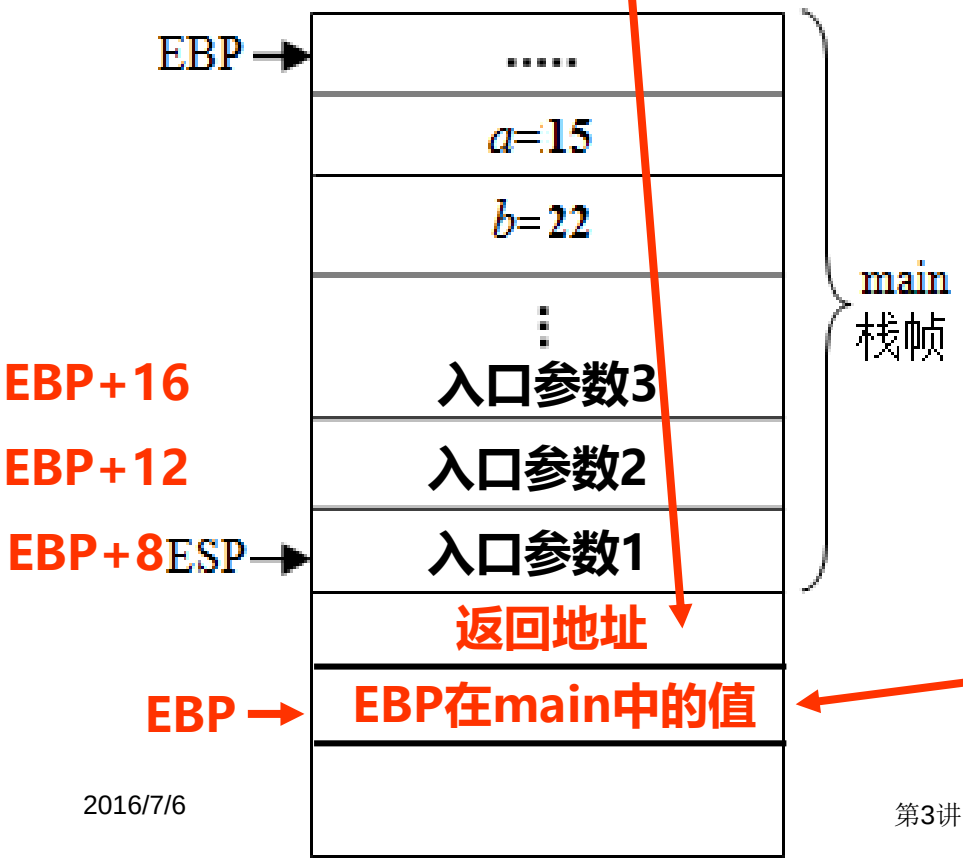
过程（函数）的结构

- 一个C过程的大致结构如下：
  - 准备阶段
    - 形成帧底：push指令 和 mov指令
    - 生成栈帧（如果需要的话）：sub指令 或 and指令
    - 保存现场（如果有被调用者保存寄存器） ：mov指令
  - 过程（函数）体
    - 分配局部变量空间，并赋值
    - 具体处理逻辑，如果遇到函数调用时
      - 准备参数：将实参送栈帧入口参数处
      - CALL指令：保存返回地址并转被调用函数
    - 在EAX中准备返回参数
  - 结束阶段
    - 退栈：leave指令 或 pop指令
    - 取返回地址返回：ret指令

入口参数的位置

```
movl  参数3，8(%esp)
……
movl  参数1, (%esp)
call   add   R[esp]←R[esp]-4
             M[R[esp]]←返回地址
             R[eip]←add函数首地址
```

准备入口参数



返回地址是什么？

call指令的下一条指令的地址！

- IA-32中，若参数类型是unsigned char、char或unsigned short、short，也都分配4个字节
- 故在被调用函数中，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
- 每个过程开始两条指令

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

过程调用参数传递举例

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

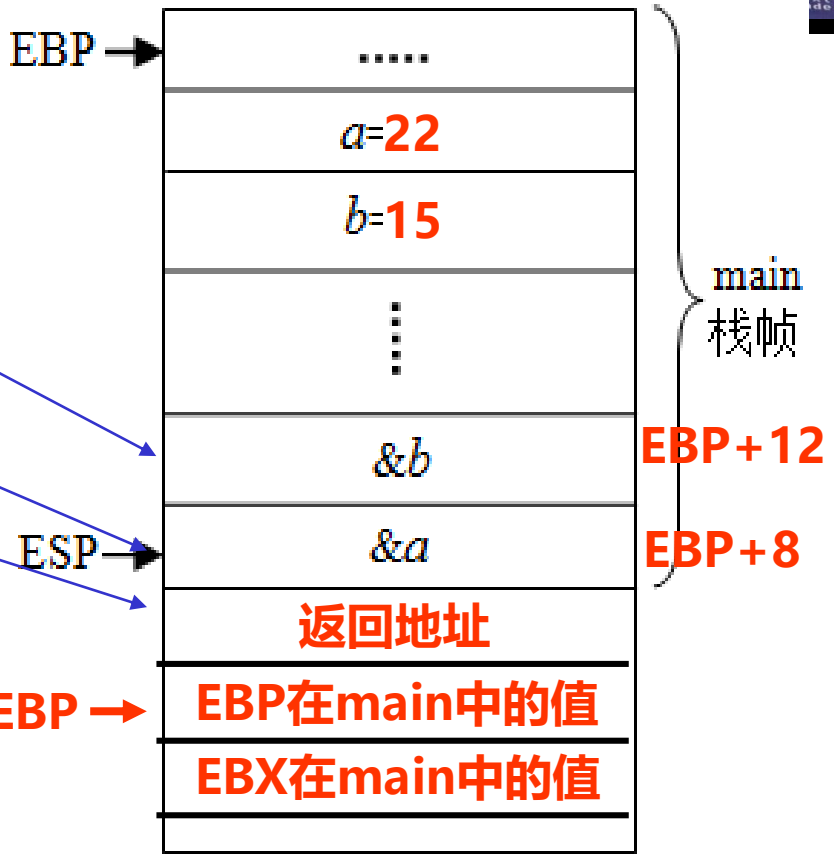
a=15　b=22

过程调用参数传递举例

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
        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 →

EBX是被调用者保存

R[ecx]←M[&a]=15

R[ebx]←M[&b]=22

M[&a] ← R[ebx] =22

M[&b] ← R[ecx] = 15

局部变量a和b进行了交换

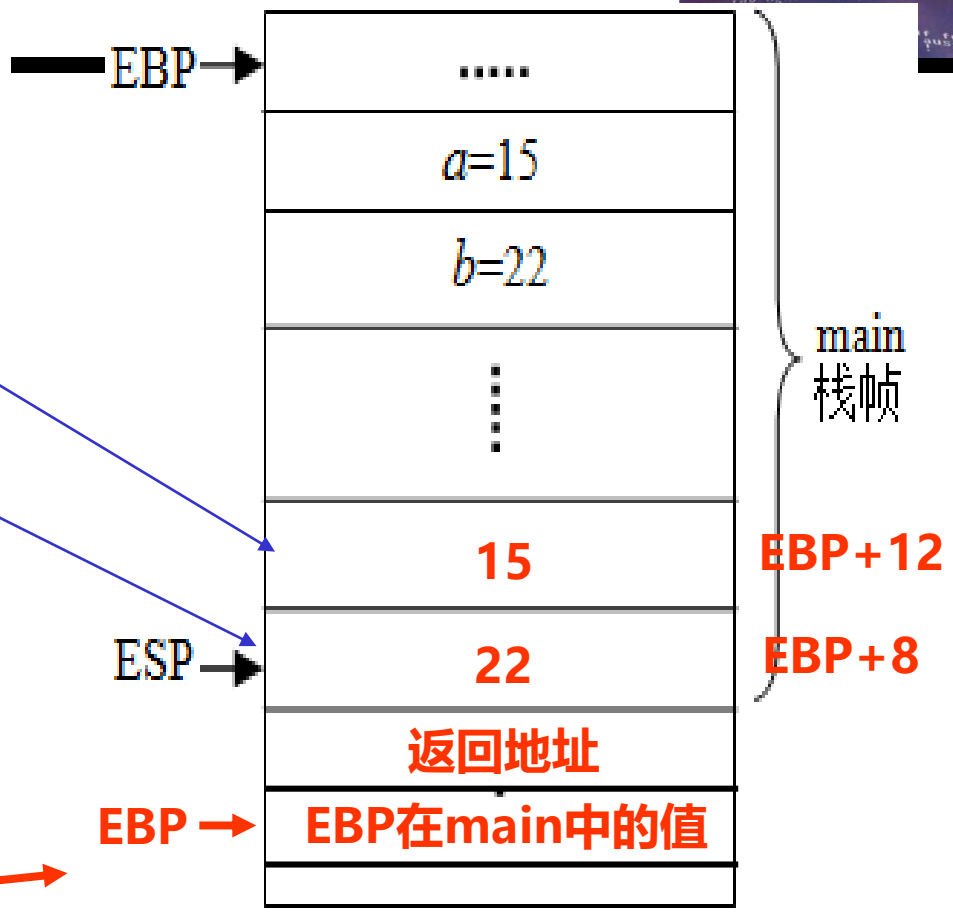
过程调用参数传递举例

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:
```

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

```
     pushl   %ebp
     movl    %esp, %ebp
     movl    8(%ebp), %edx
     movl    12(%ebp), %eax
     movl    %eax, 8(%ebp)
     movl    %edx, 12(%ebp)
```



R[edx]←15

R[eax]←22

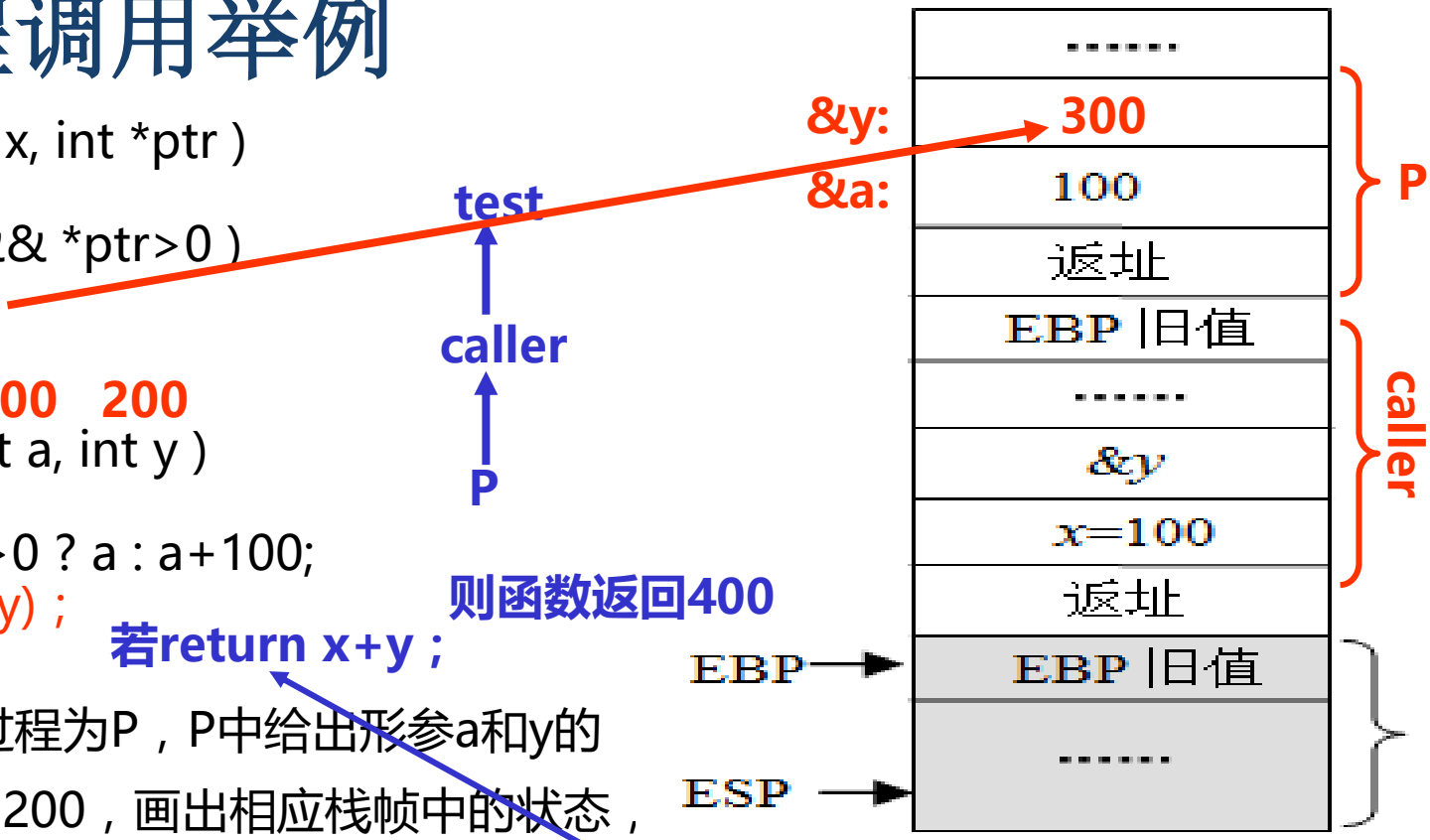
局部变量a和b没有交换，
交换的仅是入口参数

M[R[ebp]+8] ← R[eax] =22

M[R[ebp]+12] ← R[edx] =15

过程调用举例

```
1  void test ( int x, int *ptr )
2  {
3        if  ( x>0 && *ptr>0 )
4        *ptr+=x;
5   }
6                    100   200
7  void caller (int a, int y )
8  {
9         int x = a>0 ? a : a+100;
10        test (x, &y) ;
11  }
```



则函数返回400

若return x+y ;

调用caller的过程为P，P中给出形参a和y的实参分别是100和200，画出相应栈帧中的状态，

（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？ **前者按值、后者按地址。一定是一个地址**

（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200变成300，test退帧后，caller中通过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中无变量与之对应，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
    movl    $0, %eax
    cmpl    $0, %ebx
    jle     .L2
    leal    -1(%ebx), %eax
    movl    %eax, (%esp)
    call    nn_sum
    addl    %ebx, %eax
.L2
    addl    $4, %esp
    popl    %ebx
    popl    %ebp
    ret
```

nn_sum(n-1)

nn_sum(n)

P

R[ebx]←n

R[eax]←0

if (n≤0) 转L2

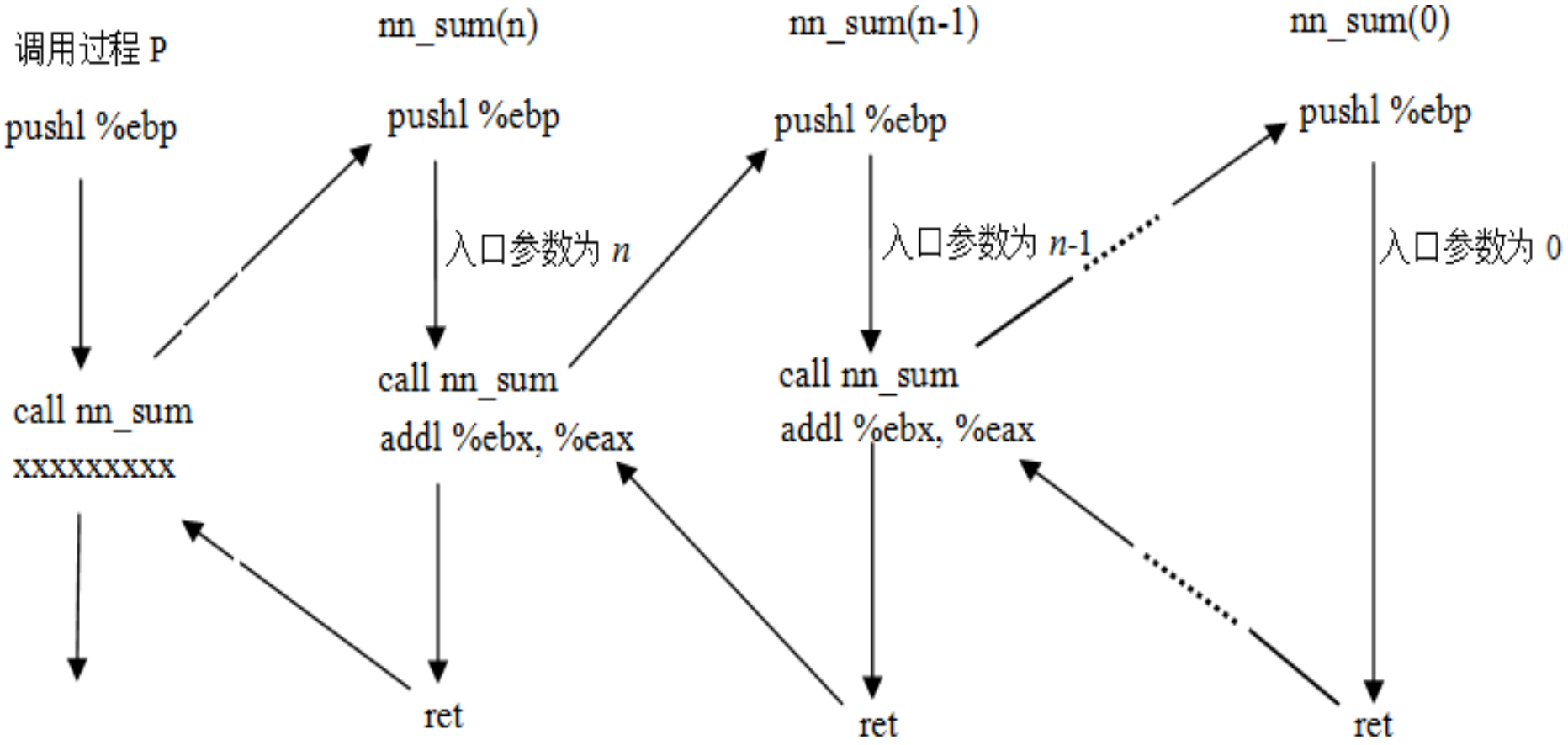
R[eax]←n-1

R[eax] ← 0+1+2+...+(n-1)+n

每次递归调用都会增加一个栈帧（该例为16B），所以空间开销很大。

过程调用的机器级表示

- 递归函数nn_sum的执行流程



为支持过程调用，每个过程包含准备阶段和结束阶段。因而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，它们对程序性能影响很大，应尽量避免不必要的过程调用，特别是递归调用。

过程调用举例

例：应始终返回d[0]中的3.14，但并非如此。Why?

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824; /* Possibly out of bounds */
  return d[0];
}
```

fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14, 然后存储保护错

不同系统上执行结果可能不同

例如，编译器对局部变量分配方式可能不同

为何每次返回不一样？

为什么会引起保护错？

栈帧中的状态如何？

```
double fun(int i)
{
  volatile double d[1] = {3.14};
  volatile long int a[2];
  a[i] = 1073741824;
  return d[0];
}
```

当i=0或1，OK
当i=2, d3~d0=0x40000000
低位部分（尾数）被改变
当i=3, d7~d3=0x40000000
高位部分被改变
当i=4, EBP被改变

```
<fun>:
    push   %ebp
    mov    %esp,%ebp
    sub    $0x10,%esp
    fldl   0x8048518
    fstpl  -0x8(%ebp)
    mov    0x8(%ebp),%eax
    movl   $0x40000000,-0x10(%ebp,%eax,4)
    fldl   -0x8(%ebp)
    leave
    ret
```

EBP	EBP的旧值	4
	d7 ... d4	3
	d3 ... d0	2
	a[1]	1
ESP	a[0]	0

a[i]=1073741824;

0x40000000
$=2^{30}=1073741824$

return d[0];

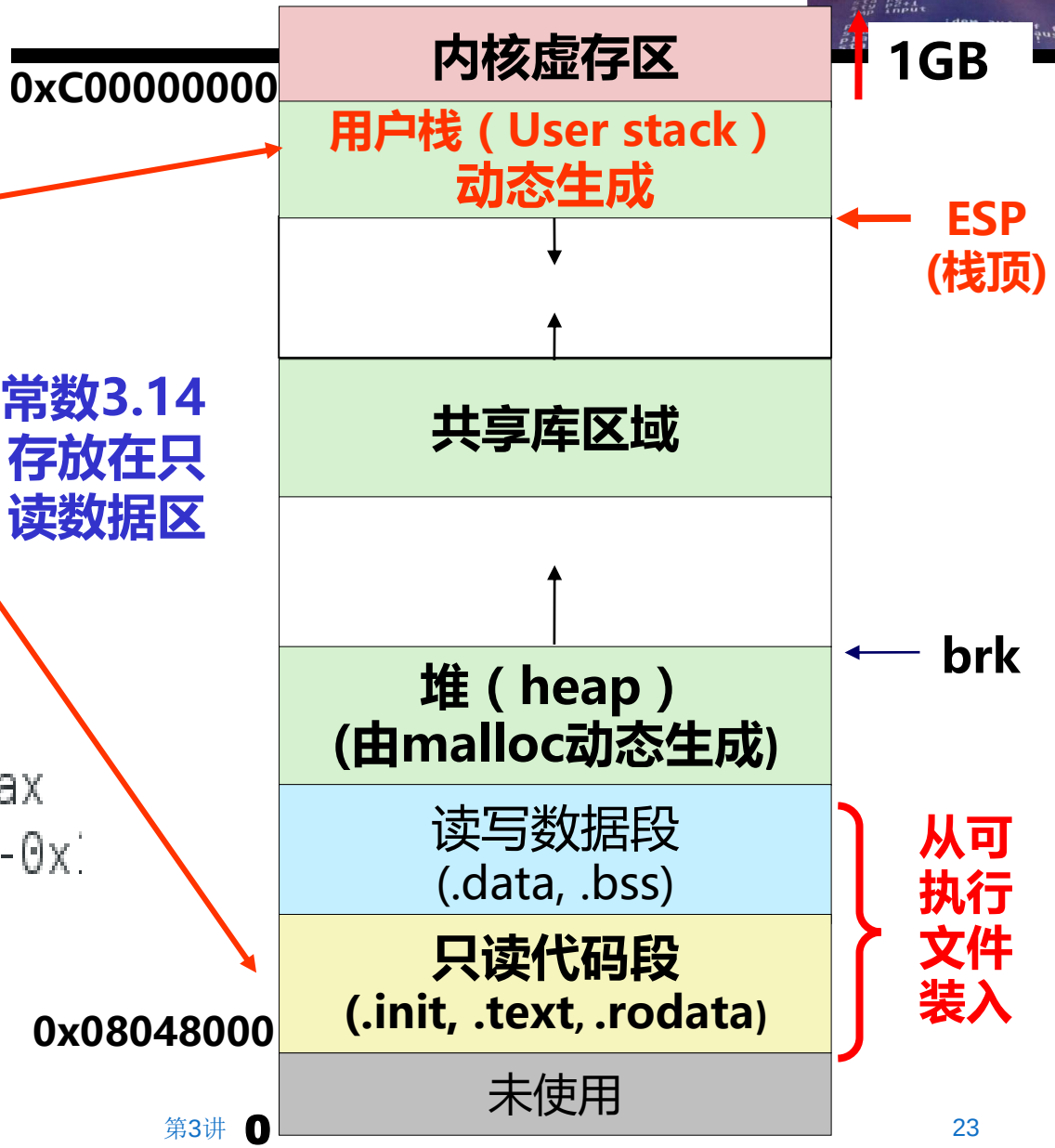
fun(2) = 3.13999998664856
fun(3) = 2.00000061035156
fun(4) = 3.14, 然后存储保护错

IA-32/Linux的存储映像

只读代码段地址的特点：
0x8048xxx
栈区地址特点：
0xbfffxxxx

```
<fun>:
    push   %ebp
    mov    %esp,%ebp
    sub    $0x10,%esp
    fldl   0x8048518
    fstpl  -0x8(%ebp)
    mov    0x8(%ebp),%eax
    movl   $0x40000000,-0x
    fldl   -0x8(%ebp)
    leave
    ret
```

常数3.14存放在只读数据区



Windows中的存储映像

```
#include  ....
int g1=0, g2=0, g3=0;
int main()
{
    static int s1=0, s2=0, s3=0;
    int v1=0, v2=0, v3=0;
    printf("0x%08x\n",&v1);
    printf("0x%08x\n",&v2);
    printf("0x%08x\n\n",&v3);
    printf("0x%08x\n",&g1);
    printf("0x%08x\n",&g2);
    printf("0x%08x\n\n",&g3);
    printf("0x%08x\n",&s1);
    printf("0x%08x\n",&s2);
    printf("0x%08x\n\n",&s3);
    return 0;
}
```

说明了什么？

注意：每个存储区地址的特征！

执行结果如下：

0x0012ff78
0x0012ff7c
0x0012ff80

局部变量存放在另一个存储区：栈区

0x004068d0
0x004068d4
0x004068d8
0x004068dc
0x004068e0
0x004068e4

全局变量和静态变量连续存放在同一个存储区：可读写数据区

Windows中的存储映像

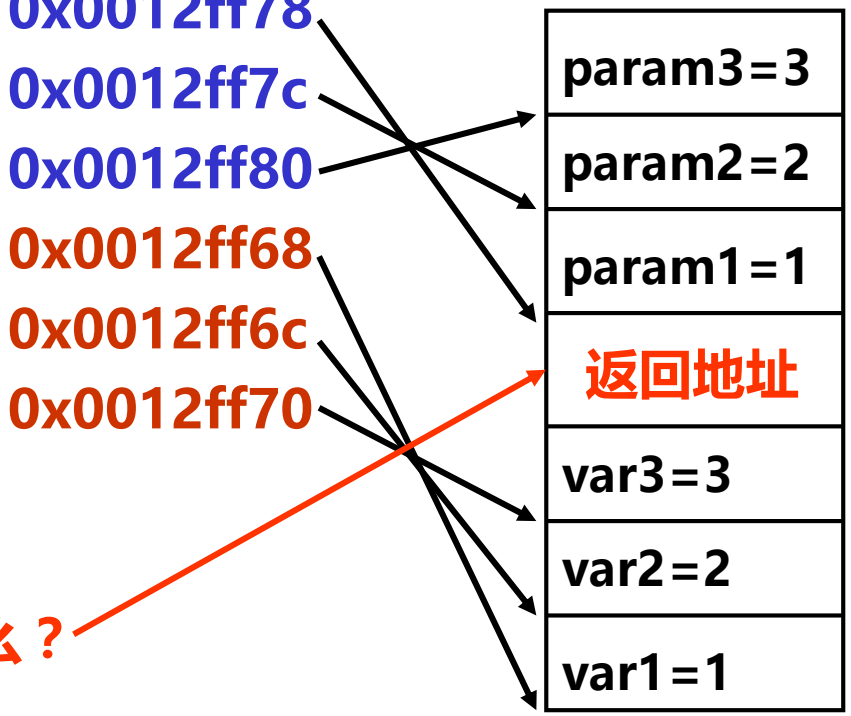
```
#include  ……
void __stdcall func(int param1,int param2,int param3)
{
    int var1=param1;
    int var2=param2;
    int var3=param3;
    printf( “0x%08x\n” ,&param1);
    printf("0x%08x\n", &param2);
    printf("0x%08x\n\n", &param3);
    printf("0x%08x\n",&var1);
    printf("0x%08x\n",&var2);
    printf("0x%08x\n\n",&var3);
    return;
}
int main()
{
    func(1,2,3);
    return 0;
}
```

说明了什么？

Windows中栈区也是高地址向低地址生长！

执行结果如下：

0x0012ff78
0x0012ff7c
0x0012ff80
0x0012ff68
0x0012ff6c
0x0012ff70



猜猜这里是什么？

这里与Linux的差别是什么？ EBP！

有关“过程调用”的练习

假设P为调用过程，Q为被调用过程，程序在IA-32处理器上执行，以下有关过程调用的叙述中，错误的是（B）。

A. C语言程序中的函数调用就是过程调用

B. 从P传到Q的实参无需重新分配空间存放

C. 从P跳转到Q执行应使用CALL指令

D. 从Q跳回到P执行应使用RET指令

有关“过程调用”的练习

假设P为调用过程，Q为被调用过程，程序在IA-32处理器上执行，以下是C语言程序中过程调用所涉及的操作：

① 过程Q保存P的现场，并为非静态局部变量分配空间

② 过程P将实参存放到Q能访问到的地方

③ 过程P将返回地址存放到特定处，并跳转到Q执行

④ 过程Q取出返回地址，并跳转回到过程P执行

⑤ 过程Q恢复P的现场，并释放局部变量所占空间

⑥ 执行过程Q的函数体

过程调用的正确执行步骤是（C）。

A. ②→③→④→①→⑤→⑥

B. ②→③→①→④→⑥→⑤

C. ②→③→①→⑥→⑤→④

D. ②→③→①→⑤→⑥→④

有关“过程调用”的练习

以下是有关IA-32/Linux（GCC）的过程调用的叙述，错误的是（A）。

A. 在过程中通常先使用被调用者保存寄存器

B. 每个非叶子过程都有一个栈帧，其大小为16B的倍数

C. EBP寄存器中的内容指向对应栈帧（stack frame）的底部

D. 每个栈帧底部单元中存放其调用过程的EBP内容

以下是有关IA-32/Linux的过程调用的叙述，错误的是（C）。

A. 每进行一次过程调用，用户栈从高地址向低地址增长出一个栈帧

B. 从被调用过程返回调用过程之前，被调用过程会释放自己的栈帧

C. 只能通过将栈指针ESP作为基址寄存器来访问用户栈中的数据

D. 过程嵌套调用深度越深，栈中栈帧个数越多，严重时会发生栈溢出

有关“过程调用”的练习

以下是有关C程序的变量作用域和生存期的叙述，错误的是（ ）。 **A**

A. 静态（static型）变量和非静态局部变量都分配在对应栈帧中

B. 因为非静态局部变量被分配在栈中，所以其作用域仅在过程体内

C. 非静态局部变量可以与全局变量同名，因为它们被分配在不同存储区

D. 不同函数中非静态局部变量可以同名，因为它们被分配在不同栈帧中

以下有关递归过程调用的叙述中，错误的是（C）。

A. 每次递归调用都会额外执行多条指令，因而时间开销大

B. 每次递归调用都会生成一个新的栈帧，因而空间开销大

C. 每次递归调用在栈帧中保存的返回地址都不相同

D. 递归过程第一个参数的有效地址为R[ebp]+8

有关“过程调用”的练习

以下是一个C语言程序代码：

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

int caller( )
{
    int t1=100 ;
    int t2=200;
    int sum=add(t1, t2);
    return sum;
}
```

以下关于上述程序代码在 IA-32上执行情况的叙述中，错误的是（C）。

A. 变量t1、t2和sum被分配在寄存器或caller函数的栈帧中

B. 传递参数时t2和t1的值从高地址到低地址依次存入栈中

C. 入口参数t1和t2的值被分配在add函数的栈帧中

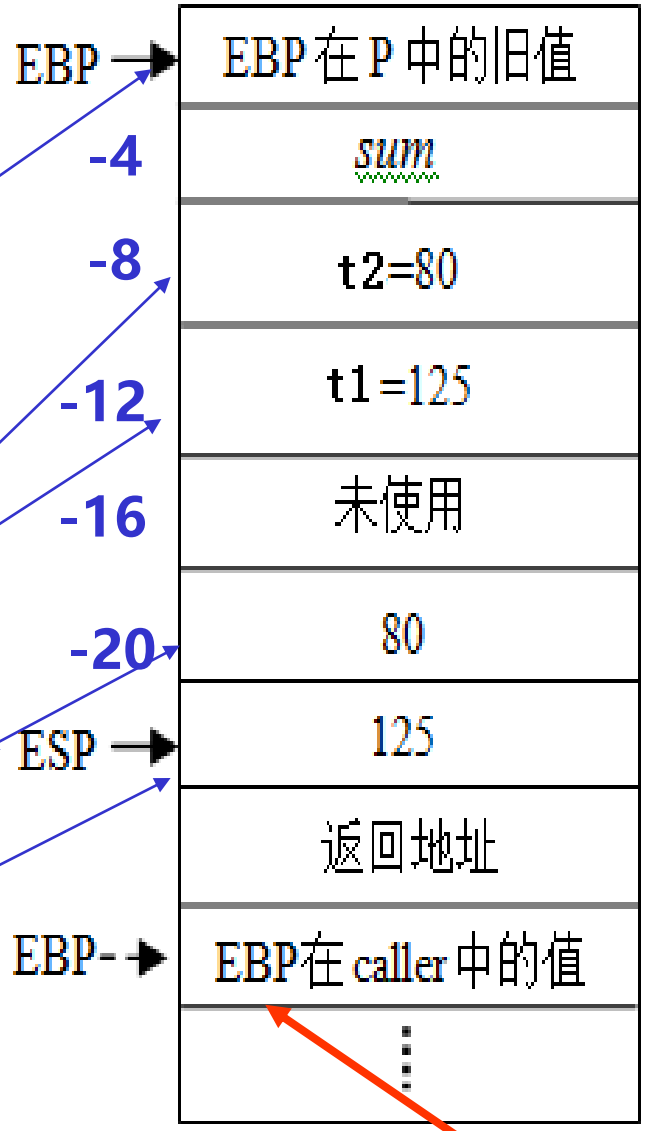
D. add函数返回时返回值存放在EAX寄存器中

```
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

caller

P



caller 帧底

caller 栈帧

ESP+4

add 栈帧底部

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

movl %ebp, %esp
popl %ebp

add函数开始是什么？

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

有关“过程调用”的练习

以下是一个C语言程序代码：

```
int add(int *xp, int *yp)
{    return *xp+*yp;  }
void caller( )
{
    static int t1=100;
    static int t2=200;
    int sum=add(&t1, &t2);
    int diff=sub(&t1, &t2);
    printf( “sum=%d, diff=%d", sum, diff);
}
```

思考题：

若改为以下语句，则怎样？

```
int diff;
sub(&diff,&t1,&t2);
```

以下关于上述代码在 IA-32/Linux上执行情况的叙述中，错误的是 (**B**) 。

A. 变量t1、t2被分配在可读可写的全局静态数据区中

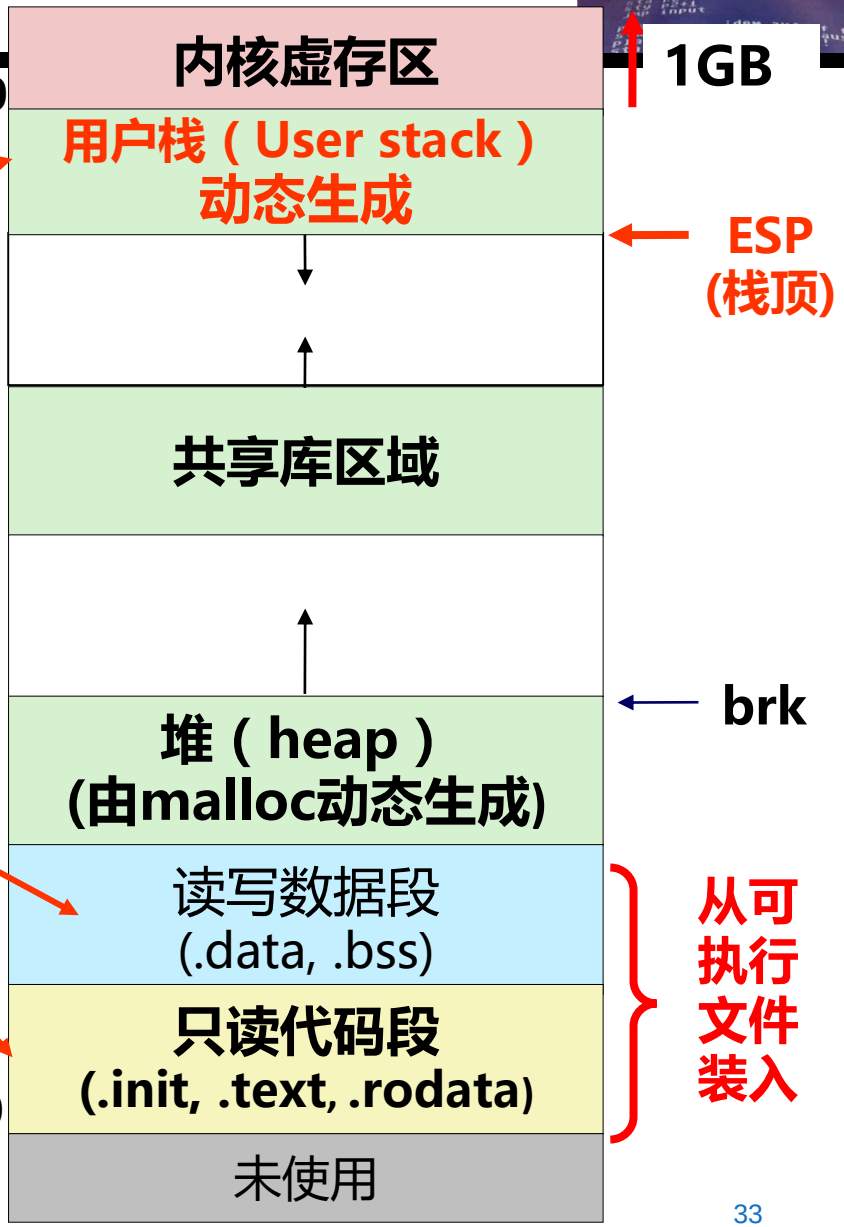
B. 存入栈中的入口参数可能是0xbfff0004、0xbfff0000

C. 在caller中执行leave指令后，入口参数的值还在存储器中

D. add函数和sub函数的栈帧底部在完全相同的位置处

IA-32/Linux的存储映像

只读代码段：
0x8048xxx
栈区：
0xbfffxxxx
全局静态数据区：
0x8049xxx



```
int add ( int x, int y ) {
    return x+y;
}
int  caller ( ) {
    ......
    int   sum = add (t1, t2);
    int  diff= sub (t1, t2);
    ......
}
```

add

caller

P

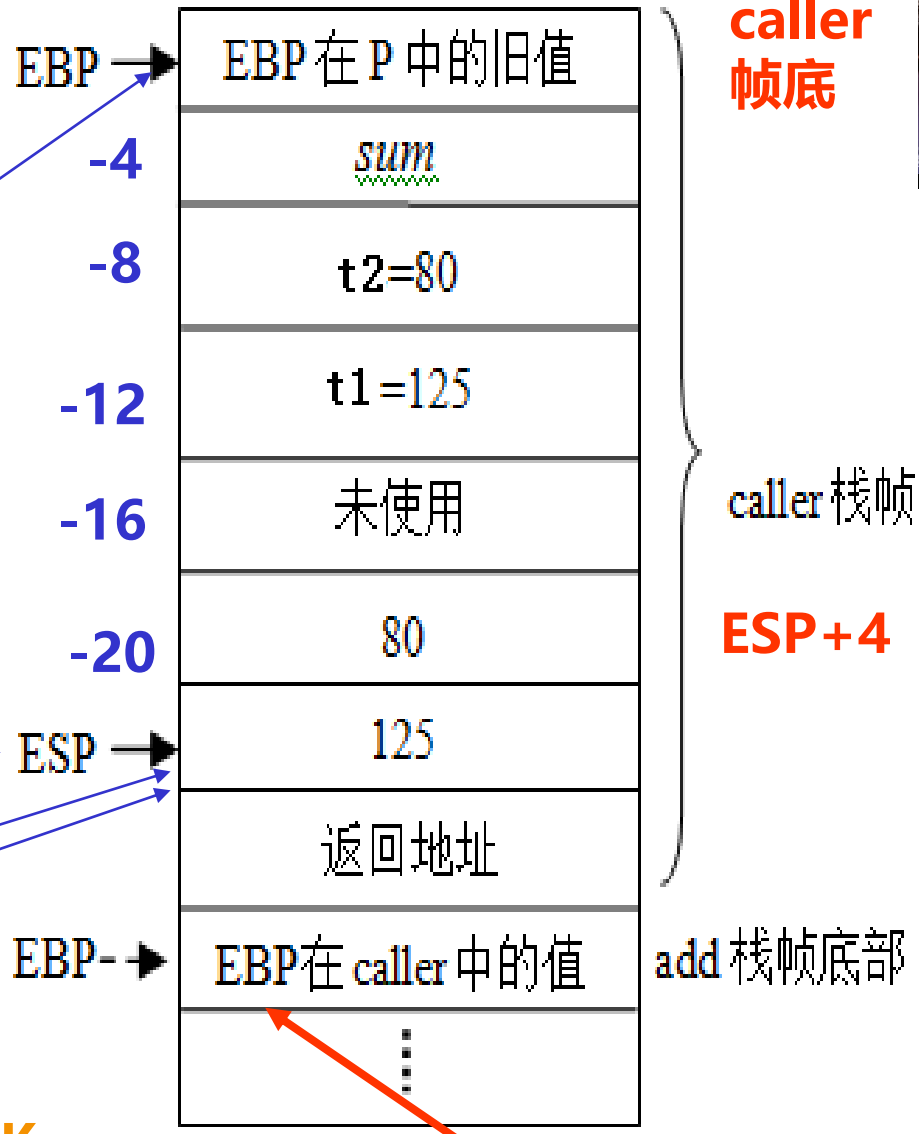
```
caller :
 pushl    %ebp
 movl     %esp, %ebp
 subl     $24, %esp
 ......
 ......
 movl     -12(%ebp), %eax
 movl     %eax, (%esp)
 call     add
 .......
 call     sub
 ......
 leave
 ret
```

准备阶段

准备入口参数

准备入口参数

结束阶段



BACK

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

add函数开始是什么？

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

有关“过程调用”的讨论

为什么以下程序输出结果是x=-1217400844而不是x=100？在你的机器上执行结果是什么？每次执行结果都一样吗？反汇编后的机器级代码如何支持你的分析？

```
int x=100 ;
void main ( )
{    int x;
     printf( “x=%d\n” , x) ;
}
```

稍作修改后输出结果是什么？

```
int x=100 ;
void main ( )
{   int x=10;
    printf( “x=%d\n” , x) ;
}
```

```
int x=100 ;
void main ( )
{     static int x;
      printf( “x=%d\n” , x) ;
}
```

```
void main ( )
{   int x;
    printf( “x=%d\n” , x) ;
}
```

字符串“x=%d\n”属于只读数据

```
0804841c <main>:
 804841c:       55                              push   %ebp
 804841d:       89 e5                           mov    %esp,%ebp
 804841f:       83 e4 f0                        and    $0xfffffff0,%esp
 8048422:       83 ec 20                        sub    $0x20,%esp
 8048425:       8b 44 24 1c                     mov    0x1c(%esp),%eax
 8048429:       89 44 24 04                     mov    %eax,0x4(%esp)
 804842d:       c7 04 24 d0 84 04 08            movl   $0x80484d0,(%esp)
 8048434:       e8 c7 fe ff ff                  call   8048300 <printf@plt>
 8048439:       c9                              leave
 804843a:       c3                              ret
```


参考答案：

（1）程序中有两个变量x，一个是全局变量x，初值为100，另一个是局部变量x，没有赋初值。这里打印出来的x的值应该是局部变量x的值，局部变量x所占的空间是栈中的4个单元，栈中存储单元的内容不会进行初始化，除非局部变量赋初值，因而局部变量x的值是一个随机数（例如，我运行该程序两次得到的结果分别是 x=-1217400844 和 x=-1217273868）。而全局变量所占空间的初值一定是确定的，要么是程序所赋予的初值，要么是0（未赋初值时）。

（2）main函数反汇编后的结果如下，这里局部变量x所占空间的首地址为R[esp]+0x1c，没有任何一条指令对该空间的4个字节赋值，而是直接将4个字节取出，作为printf()函数的参数，存入了首地址为R[esp]+4的空间。

有关“过程调用”的讨论

- 以下是网上的一个帖子，请将程序的可执行文件反汇编（基于IA-32），并对汇编代码进行分析以正确回答该贴中的问题。

  该贴给出的结果是在Linux还是Windows上得到的？

C/C++ code

```
1  #include "stdafx.h"
2  int main(int argc, char* argv[])
3  {
4      int a=10;
5      double *p=(double*)&a;
6      printf("%f\n",*p);              //结果为0.000000
7      printf("%f\n",(double(a)));     //结果为10.000000
8
9      return 0;
10 }
11 为什么printf("%f",*p)和printf("%f",(double)a)结果不一样呢？
```

不都是强制类型转换吗？怎么会不一样

有关“过程调用”的讨论

Windows下结果如何？
p: 0x0012ffxx
a: 0xa

在32位Linux系统中反汇编结果：

```
int a = 10;
 8048425:  c7 44 24 28 0a 00 00 00    movl   $0xa,0x28(%esp)
```

+2c	p : &a=0xbfff0028
+28	a : 0xa
	.....
+8	p : &a=0xbfff0028
+4	a : 0xa
ESP	0x8048500 (指向“%f\n”的指针)

假定R[esp]=0xbfff0000

打印出来的是0!

```
 lea    0x28(%esp),%eax
 mov    %eax,0x2c(%esp)
```

编层次并没有体现出来，都是直接 mov 过去

```
 mov    0x2c(%esp),%eax
 fldl   (%eax)
```

精度加载到浮点栈顶 ST(0)）

```
 fstpl  0x4(%esp)
```

（*p 的类型是 double，故按 64 位压栈）

```
 movl   $0x8048500,(%esp)
 call   8048300 <printf@plt>
```

打印出来的是一个负数

```
 mov    0x28(%esp),%eax
 mov    %eax,0x1c(%esp)
```

由于没有优化，这里有一些冗余的 mov 操作，把变量 a 的值移来移去

```
 8048453:  db 44 24 1c
 fildl  0x1c(%esp)
```

把 10 转换成 double 型，注意这里用的是 fildl 指令，和上面用的 fldl 指令不一样!

有关“过程调用”的讨论

- 从上述代码可以看出，对于double *p=(double *)&a，只是把a的地址直接传送到p所存放的空间，然后把p中的内容，也就是a的地址送到了EAX中，随后用指令“fldl (%eax)”将a的地址处开始的8个字节的机器数（xx…x0000000AH）直接加载到ST(0)中，其中前4个字节xx…x表示R[esp]+0x28，在Linux系统中它应该是一个很大的数，如BFFF…，然后再用指令“fstpl 0x4(%esp)”把ST(0)中的内容（即xx…x0000000AH）作为printf函数的参数送到R[esp]+4的位置，printf(“%lf\n,*p)函数将其作为double类型（%lf）的数打印出来。显然，这个打印的值不会是10.000000，而是一个负数。
- 因为Linux和Windows两种系统所设置的栈底所在地址不同，所以ESP寄存器中的内容不同，因而打印出来的值也肯定不同。通常，Linux中栈底在靠近C0000000H的位置，而在Windows中栈的大致位置是0012FFxxH。因此，可以判断出题目中给出的结果应该是在Windows中执行的结果，打印的值应该是0012 FFxx 0000 000AH 或者 0000 000A 0012 FFxxH对应的double类型的值，前者值为+1.0010….1010×2^{-1022}，后者为+0.0…1… ×2^{-1023}，显然都是接近0的值，正如题目中程序注释所示，结果为0.000000。
- 对于printf(“%lf\n, (double)a) 函数，使用的指令为fildl，该指令先将a作为int型变量（值为10）等值转换为double类型，再加载到ST(0)中。这样再作为double类型（%lf）的数打印时，打印的值就是10.000000。

有关“过程调用”的讨论

例：以下是一段C语言代码：

```
#include <stdio.h>
main()
{
    double a = 10;
    printf("a = %d\n", a);
}
```

在IA-32上运行时，打印结果为a=0

在x86-64上运行时，打印一个不确定值

为什么？

10=1010B=1.01×2^3

阶码e=1023+3=10000000010B

10的double型表示为：

0 10000000010 0100…0B

即4024 0000 0000 0000H

先执行fldl，再执行fstpl

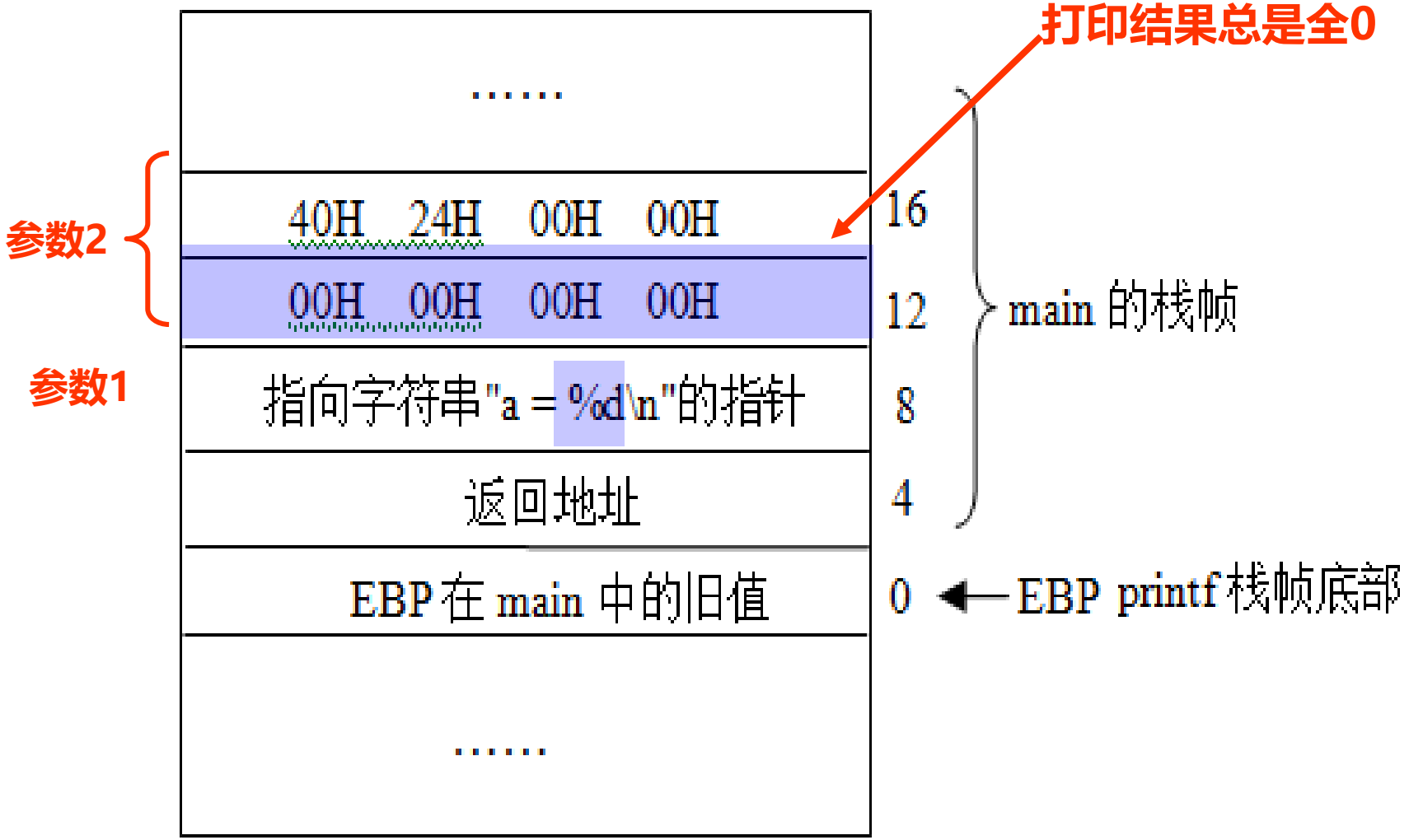
fldl：局部变量区→ST(0)

fstpl：ST(0) →参数区

在IA-32中a为float型又怎样呢？先执行flds，再执行fstpl

即：flds将32位单精度转换为80位格式入浮点寄存器栈，fstpl再将80位转换为64位送存储器栈中，故实际上与a是double效果一样！

IA-32过程调用参数传递



a的机器数对应十六进制为：40 24 00 00 00 00 00 00H

X86-64过程调用参数传递

```
main()
{
    double a = 10;
    printf("a = %d\n", a);
}
```

操作数宽度（字节）	入口参数						返回参数
	1	2	3	4	5	6	
8	RDI	RSI	RDX	RCX	R8	R9	RAX
4	EDI	ESI	EDX	ECX	R8D	R9D	EAX
2	DI	SI	DX	CX	R8W	R9W	AX
1	DIL	SIL	DL	CL	R8B	R9B	AL

```
.LC1:
        .string "a = %d\n "
......
movsd   .LC0(%rip), %xmm0 //a送xmm0
movl    $.LC1, %edi //RDI 高32位为0
movl    $1, %eax    //向量寄存器个数
call     printf
addq    $8, %rsp
ret
......
.LC0:
    .long   0          <-- 00000000H
    .long   1076101120 <-- 40240000H
```

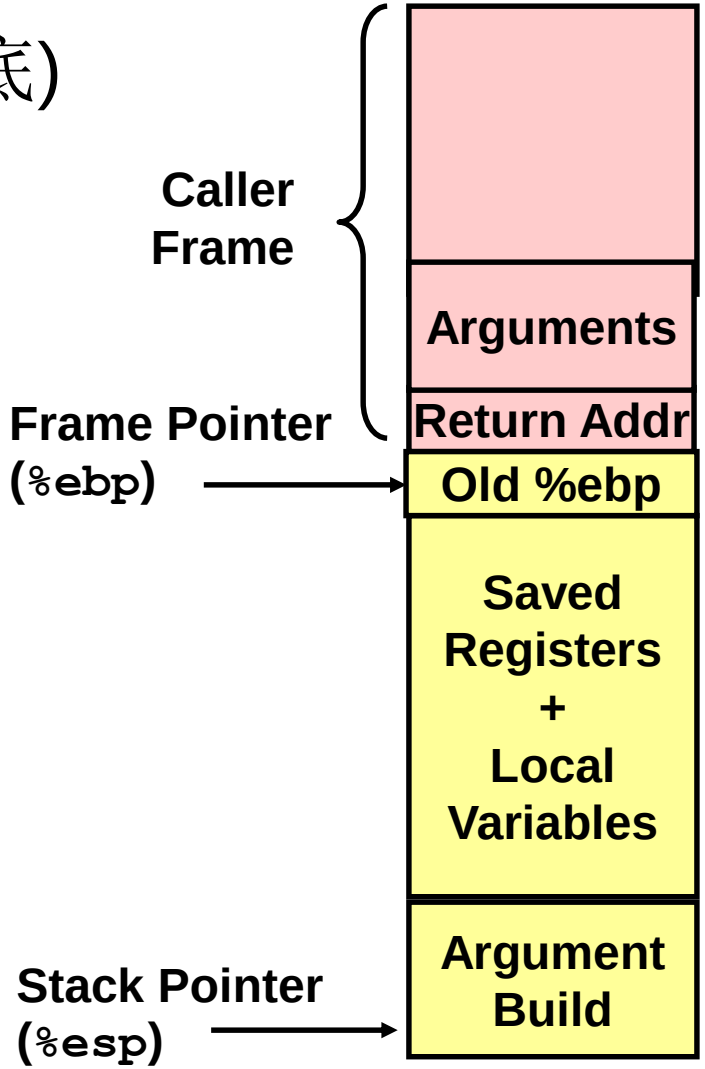
因为printf第2个参数为double型，故向量寄存器个数为1

小端方式！0存在低地址上

printf中为%d，故将从ESI中取打印参数进行处理；但a是double型数据，在x86-64中，a的值被送到XMM寄存器中而不会送到ESI中。故在printf执行时，从ESI中读取的并不是a的低32位，而是一个不确定的值。

IA32/Linux 栈帧

- 当前的栈帧Frame (从顶到底)
  - 将要调用函数的参数
    - “参数建立”
  - 局部变量
    - 如果不能保存在寄存器中
  - 保存的寄存器内容
  - 老的帧指针
- 调用者栈帧
  - 返回地址
    - 通过call指令进栈
  - 此次调用参数



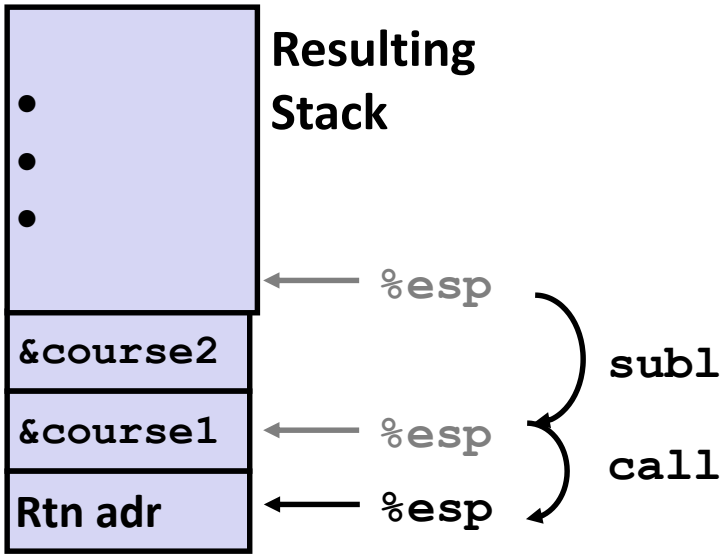
重新来看swap

```
int course1 = 15213;
int course2 = 18243;

void call_swap() {
  swap(&course1, &course2);
}
```

Calling swap from call_swap

```
call_swap:
    • • •
    subl    $8, %esp
    movl    $course2, 4(%esp)
    movl    $course1, (%esp)
    call    swap
    • • •
```

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



重新来看swap

```
void swap(int *xp,
int *yp)
{
   int t0 = *xp;
   int t1 = *yp;
   *xp = t1;
   *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx

    movl  8(%ebp), %edx
    movl  12(%ebp), %ecx
    movl  (%edx), %ebx
    movl  (%ecx), %eax
    movl  %eax, (%edx)
    movl  %ebx, (%ecx)

    popl  %ebx
    popl  %ebp
    ret
```

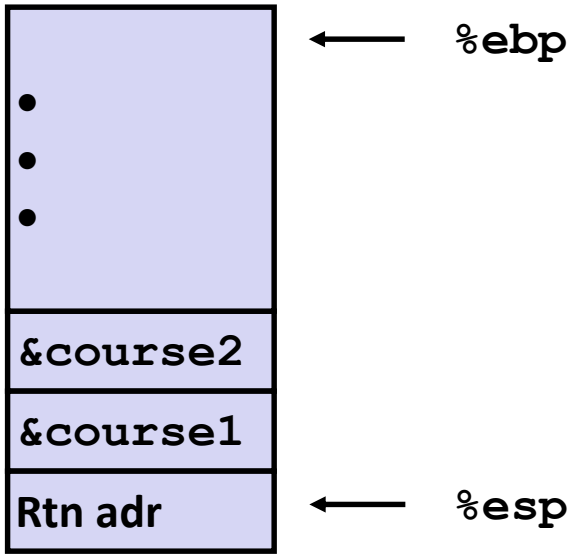
Set Up

Body

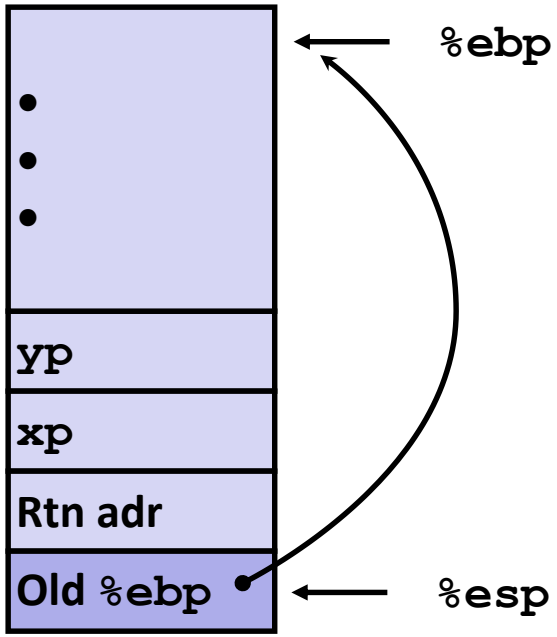
Finish

swap Setup #1

Entering Stack



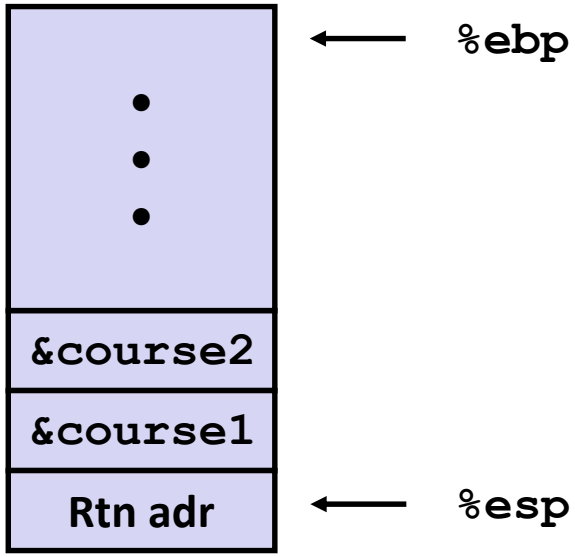
Resulting Stack



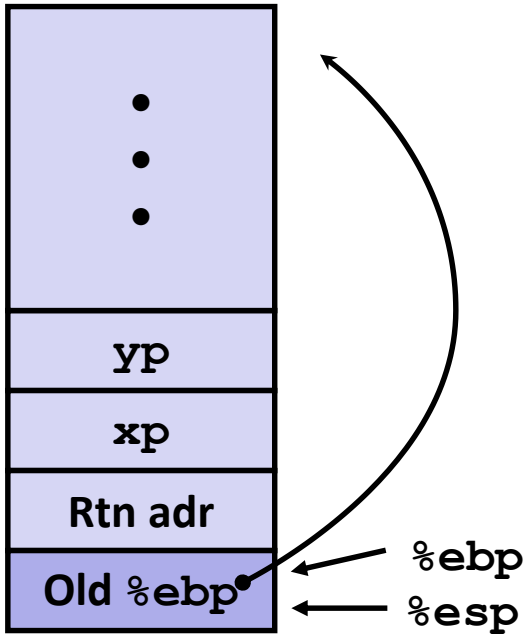

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #2

Entering Stack



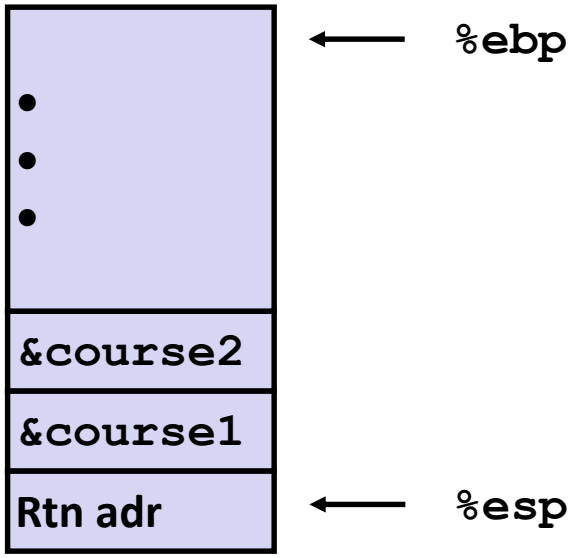
Resulting Stack



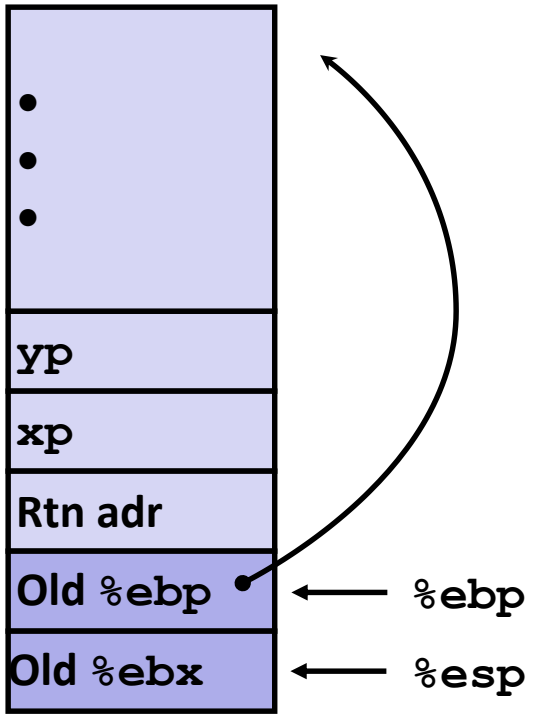

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #3

Entering Stack

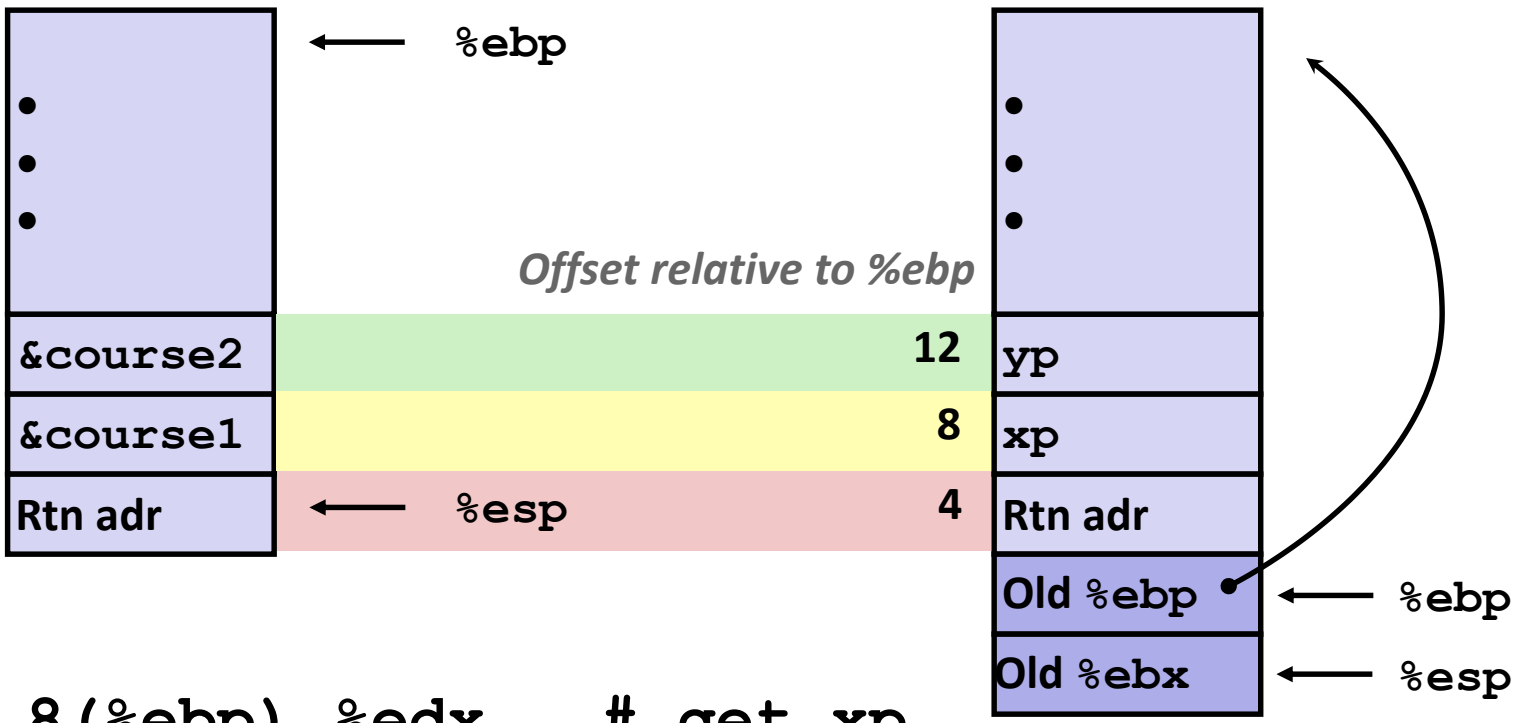


Resulting Stack




```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

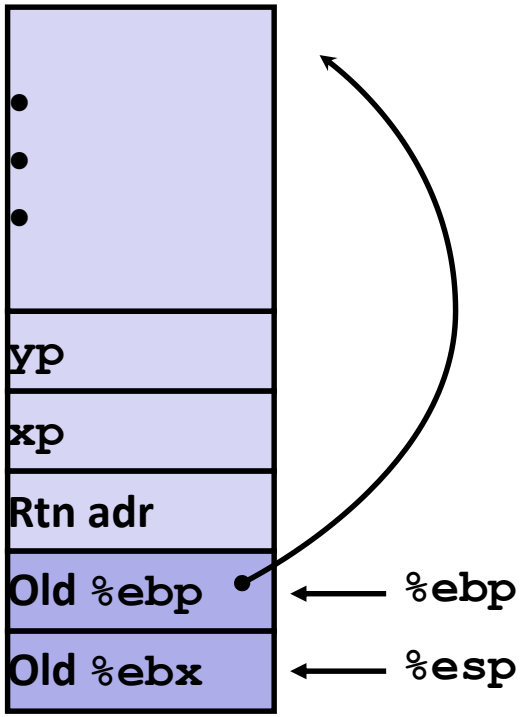
swap Body




```
movl 8(%ebp),%edx    # get xp
movl 12(%ebp),%ecx   # get yp
. . .
```

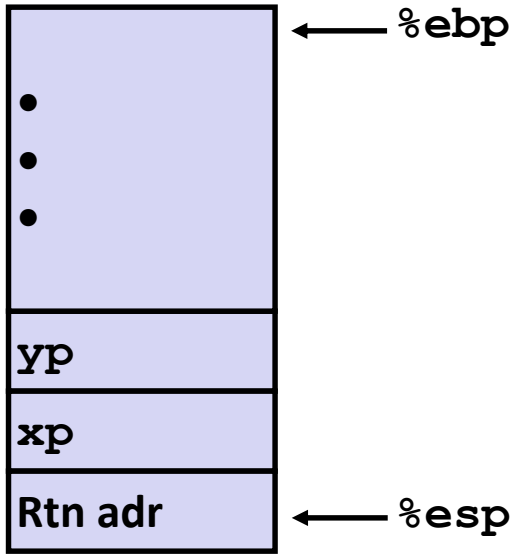
swap Finish

Stack Before Finish




```
popl    %ebx
popl    %ebp
```

Resulting Stack



- 保存&恢复寄存器 %ebx
- 对于寄存器%eax, %ecx, %edx不这样做

Disassembled swap

```
08048384 <swap>:
 8048384:  55                    push   %ebp
 8048385:  89 e5                 mov    %esp,%ebp
 8048387:  53                    push   %ebx
 8048388:  8b 55 08              mov    0x8(%ebp),%edx
 804838b:  8b 4d 0c              mov    0xc(%ebp),%ecx
 804838e:  8b 1a                 mov    (%edx),%ebx
 8048390:  8b 01                 mov    (%ecx),%eax
 8048392:  89 02                 mov    %eax,(%edx)
 8048394:  89 19                 mov    %ebx,(%ecx)
 8048396:  5b                    pop    %ebx
 8048397:  5d                    pop    %ebp
 8048398:  c3                    ret
```

Calling Code

```
 80483b4:  movl   $0x8049658,0x4(%esp) # Copy &course2
 80483bc:  movl   $0x8049654,(%esp)    # Copy &course1
 80483c3:  call   8048384 <swap>       # Call swap
 80483c8:  leave                       # Prepare to return
 80483c9:  ret                         # Return
```

寄存器保存惯例

- 当程序yoo调用who:
  - yoo 是调用者, who是被调用者
- 寄存器能用来作为临时存储空间吗?

```
yoo:
    • • •
    movl $15213, %edx
    call who
    addl %edx, %eax
    • • •
    ret
```

```
who:
    • • •
    movl 8(%ebp), %edx
    addl $91125, %edx
    • • •
    ret
```

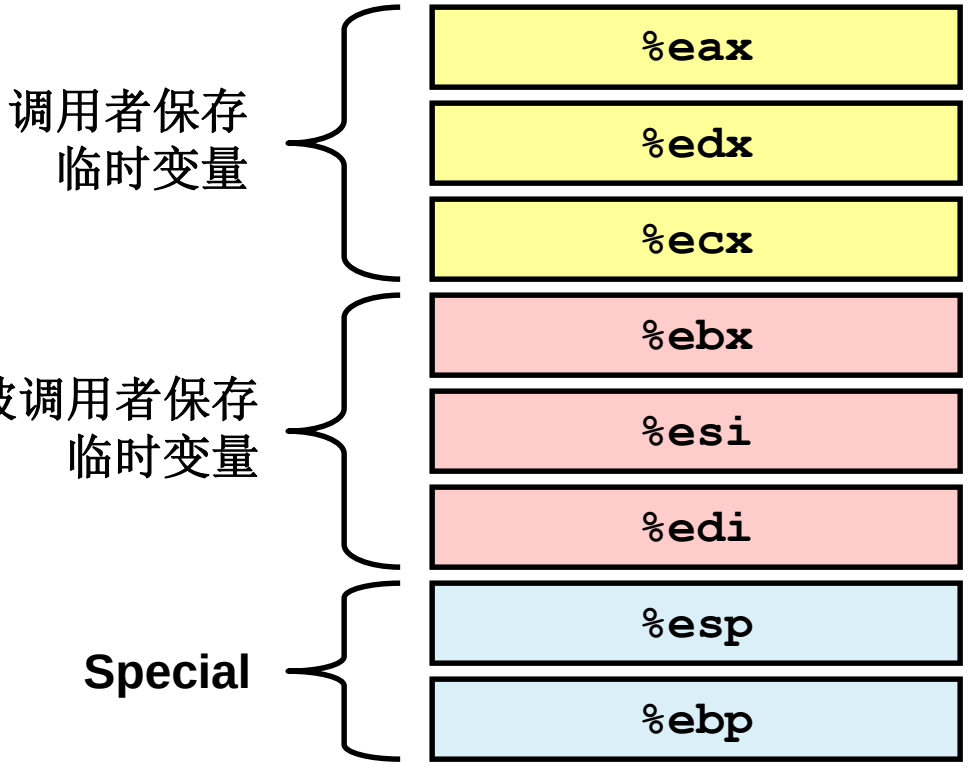
- 寄存器%edx 的内容被who改写了

寄存器保存惯例

- 当程序yoo调用who:
  - yoo是调用者，who是被调用者
- 寄存器能用来作为临时存储空间吗?
- 惯例
  - “调用者保存”
    - 调用者在调用前把临时变量保存在自己的帧中
  - “被调用者保存”
    - 被调用者在使用临时变量前先把它保存在自己的帧中

IA32/Linux 寄存器使用

- 整数寄存器
  - %ebp, %esp：被调用者保存特殊格式
    - 退出过程后恢复原始值
  - %ebx, %esi, %edi：被调用者在使用前先保存
    - 使用前将旧值保存在栈中
  - %eax, %edx, %ecx：调用者在使用前先保存
    - 可以任意修改，任何被调用者都可以修改
  - %eax：存储返回值



递归函数

```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

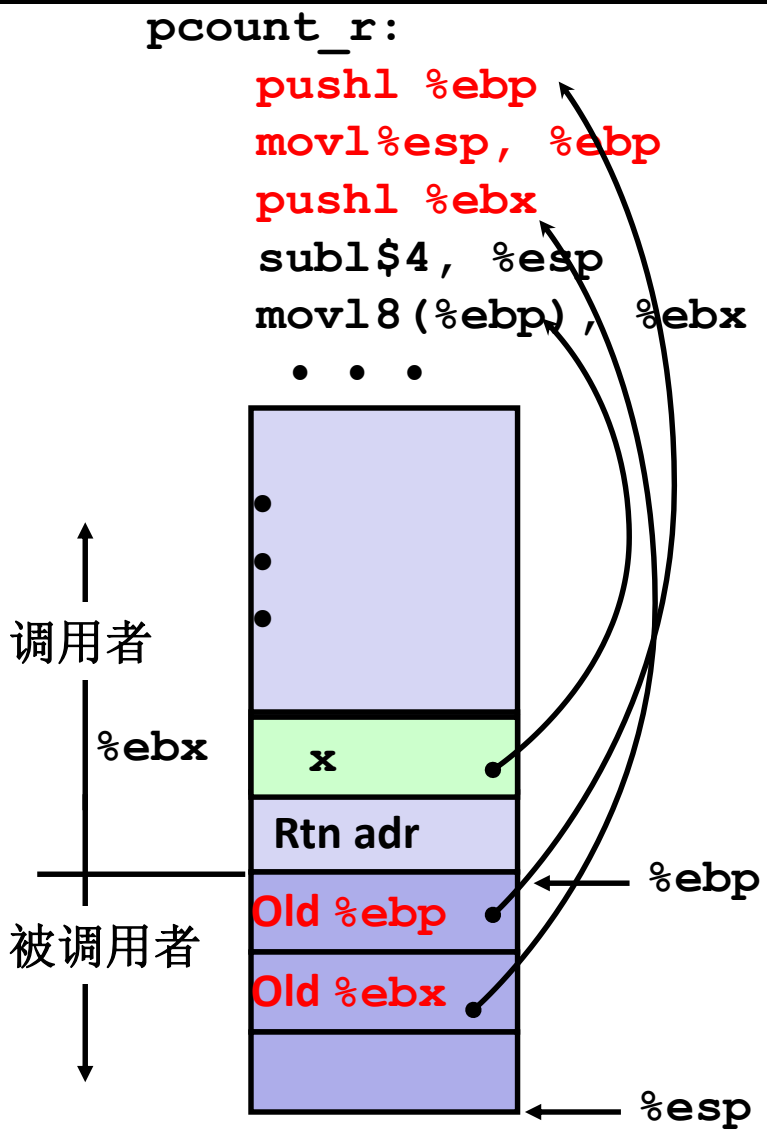
- 寄存器
  - **%eax, %edx**：使用前没有保存旧值
  - **%ebx**：使用前保存旧值，使用后恢复。

```
pcount_r:
    pushl %ebp
    movl%esp, %ebp
    pushl %ebx
    subl$4, %esp
    movl8(%ebp), %ebx
    movl$0, %eax
    testl %ebx, %ebx
    je  .L3
    movl%ebx, %eax
    shrl%eax
    movl%eax, (%esp)
    callpcount_r
    movl%ebx, %edx
    andl$1, %edx
    leal(%edx,%eax), %eax
.L3:
    addl$4, %esp
    popl%ebx
    popl%ebp
    ret
```

递归调用第1步

```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

- 动作
  - 保存 **%ebx**寄存器 中的数据到堆栈
  - 给递归调用中的变量分配空间
  - 把自变量x 值存储到 **%ebx**寄存器



递归调用第2步

```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >>
1);
}
```

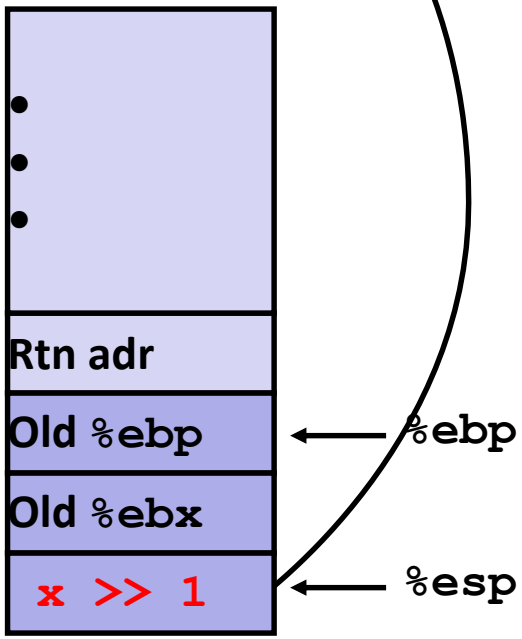
```
    • • •
   movl  $0, %eax
   testl %ebx, %ebx
   je .L3
    • • •
.L3:
    • • •
   ret
```

- 动作
  - If x == 0, return
    - 赋值 0到寄存器 **%eax** 中

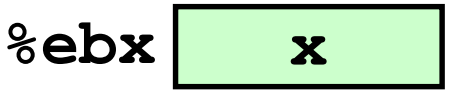
递归调用第3步

```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

```
  • • •
movl    %ebx, %eax
shrl    %eax
movl    %eax, (%esp)
call    pcount_r
  • • •
```



- 动作
  - 存储 x >> 1 后的值到堆栈
  - 执行递归调用
- 作用
  - **%eax**：保存函数结果
  - **%ebx**：保存变量 x的值

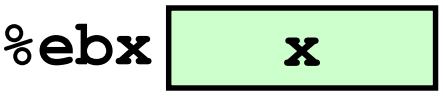


递归调用第4步

```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

```
  • • •
movl   %ebx, %edx
andl   $1, %edx
leal   (%edx,%eax), %eax
  • • •
```

- 假设
  - **%eax** 保持递归调用的返回值
  - **%ebx** 保持变量 x值
- 动作
  - 计算 (x & 1) + 加上函数返回值
- 作用
  - **%eax** 保存函数结果



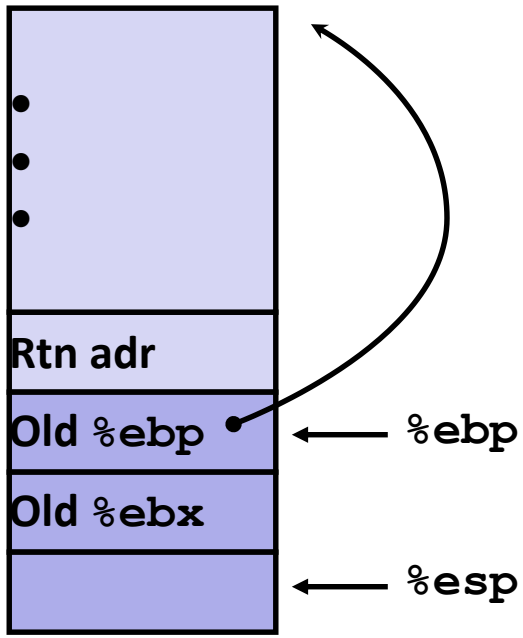
递归调用第5步

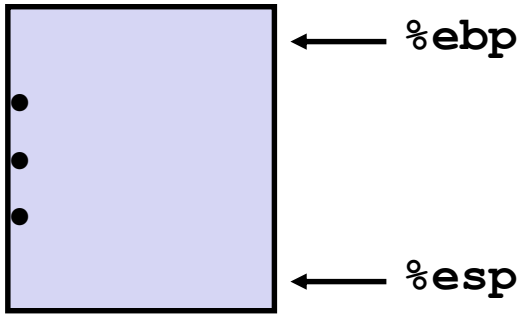
```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

```
    • • •
L3:
    addl $4, %esp
    popl %ebx
    popl %ebp
    ret
```

- 动作
  - 恢复%ebx、%ebp寄存器的数值
  - 还原%esp

%ebx | Old %ebx





递归总结

- 处置没有特别的条件
  - 每个函数调用都有私有的堆栈架构
    - 保存寄存器和本地变量
    - 保存返回指针
  - 保存寄存器的目的是防止一个函数调用被另一个函数破坏的数据
  - 堆栈体系遵循调用/返回模式
    - If P calls Q, then Q returns before P
    - 后进先出
- 支持相互递归
  - P calls Q; Q calls P

指针代码

Generating Pointer

```
/* Compute x + 3 */
int add3(int x) {
   int localx = x;
   incrk(&localx, 3);
   return localx;
}
```

Referencing Pointer

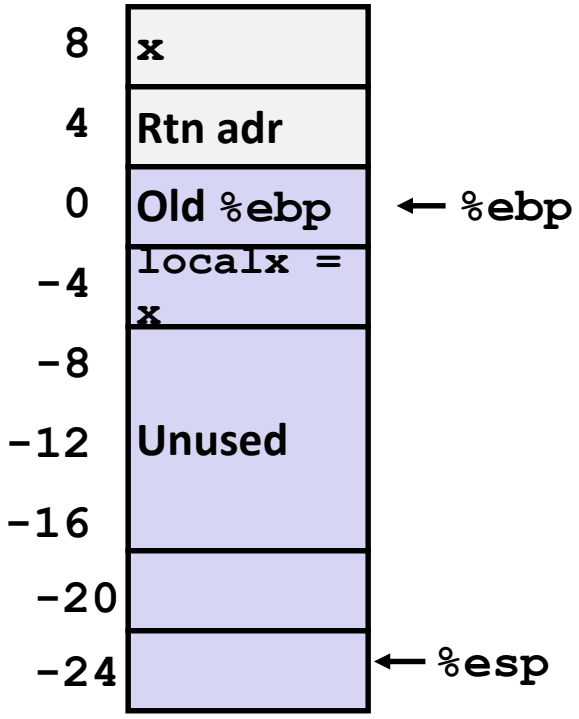
```
/* Increment value by
k */
void incrk(int *ip,
int k) {
   *ip += k;
}
```

- **Add3函数建立指针，并通过incrk函数赋值 incrk**

创建 & 初始化指针

Add3函数代码初始部分

```
add3:
   pushl %ebp
   movl  %esp, %ebp
   subl  $24, %esp       # Alloc. 24 bytes
   movl  8(%ebp), %eax
   movl  %eax, -4(%ebp) # Set localx to x
```



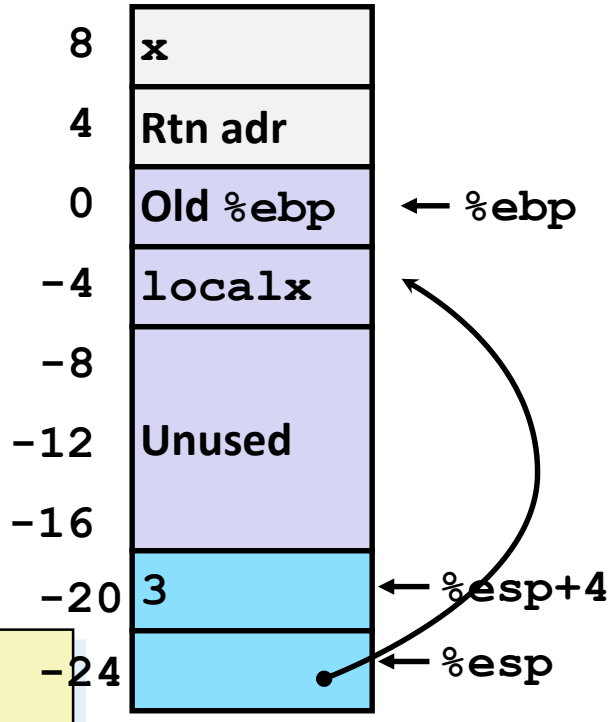
- 用栈来保存局部变量
  - 变量 val必须要存于栈
    - 需要创建指针指向它
  - 计算指针-4(%ebp)
  - 作为第二个参数进栈

```
int add3(int x) {
   int localx = x;
   incrk(&localx, 3);
   return localx;
}
```

建立指针

Add3函数代码中间部分

```
movl $3, 4(%esp)   # 2nd arg = 3
leal -4(%ebp),%eax # &localx
movl %eax, (%esp)  # 1st arg = &localx
call incrk
```



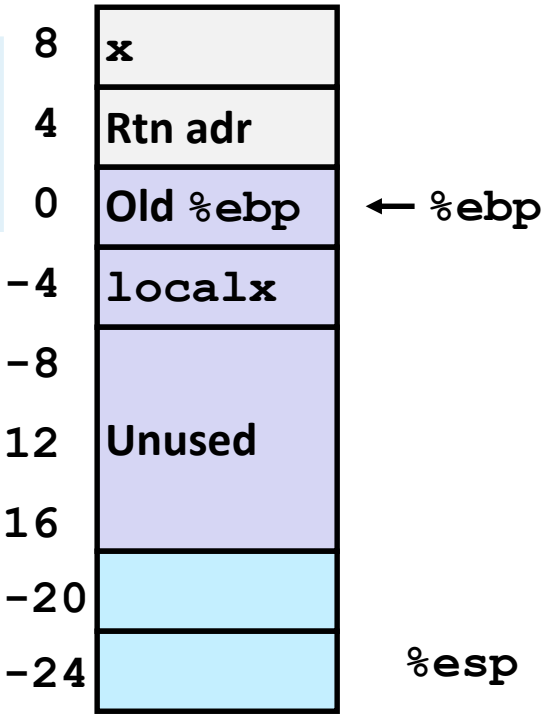
- 通过leal指令变量localx的地址

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

恢复局部变量

Add3函数代码最后部分

```
movl -4(%ebp), %eax # Return val= localx
leave
ret
```



- 从堆栈中返回localx数值

```
int add3(int x) {
   int localx = x;
   incrk(&localx, 3);
   return localx;
}
```

循环结构与递归的比较

递归函数nn_sum仅为说明原理，实际上可直接用公式，为说明循环的机器级表示，这里用循环实现。

```
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
```

局部变量 i 和 result 被分别分配在EDX和EAX中。

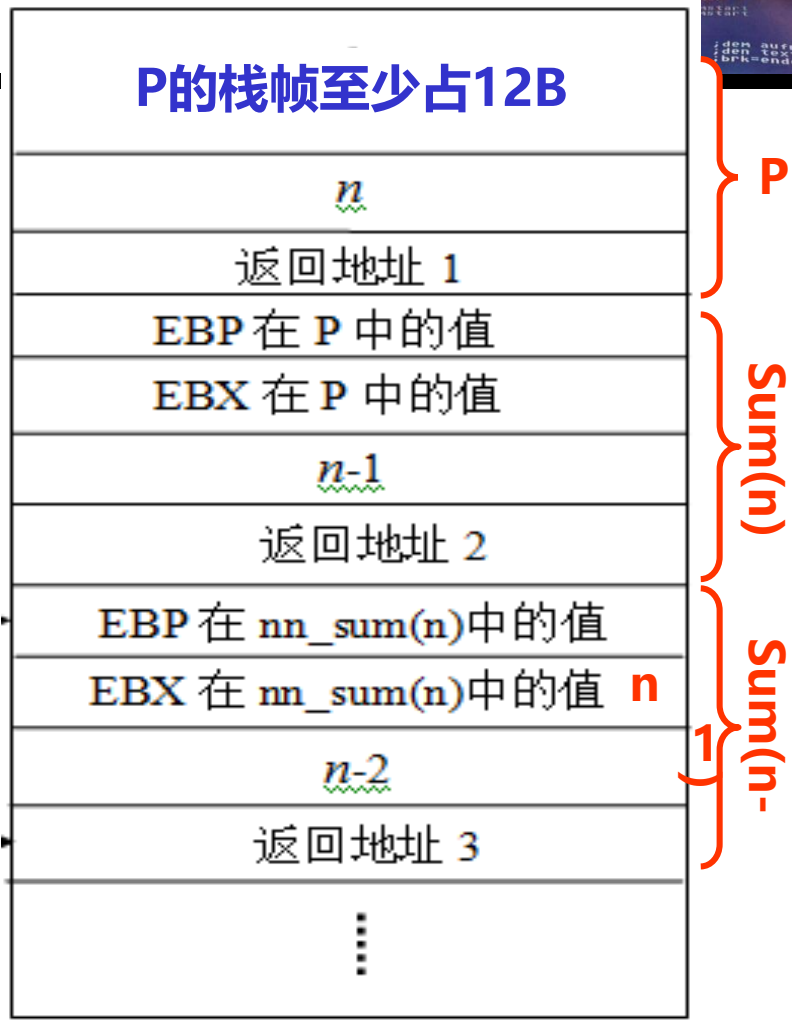
通常复杂局部变量被分配在栈中，而这里都是简单变量

过程体中没用到被调用过程保存寄存器。因而，该过程栈帧中仅需保留EBP，即其栈帧仅占用4字节空间，而递归方式则占用了(16n+12)字节栈空间，多用了(16n+8)字节，每次递归调用都要执行16条指令，一共多了n次过程调用，因而，递归方式比循环方式至少多执行了16n条指令。由此可以看出，为了提高程序的性能，若能用非递归方式执行则最好用非递归方式。

递归过程调用举例

```
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
 movl    $0, %eax
 cmpl    $0, %ebx
 jle     .L2
 leal    -1(%ebx), %eax
 movl    %eax, (%esp)
 call    nn_sum
 addl    %ebx, %eax
.L2
 addl    $4, %esp
 popl    %ebx
 popl    %ebp
 ret
```



时间开销：每次递归执行16条指令，共16n条指令

空间开销：一次调用增加16B栈帧，共16n+12

逆向工程举例

```
int function_test( unsigned x)
{
    int result=0;
    int i ;
    for ( ____①____ ; ____②____ ; ____③____ ) {
            ____________④____________ ;
    }
    return result;
}
```

```
movl   8(%ebp), %ebx
movl   $0, %eax
movl   $0, %ecx
.L12:
leal   (%eax,%eax), %edx
movl   %ebx, %eax
andl   $1, %eax
orl    %edx, %eax
shrl   %ebx
addl   $1, %ecx
cmpl   $32, %ecx
jne    .L12
```

该函数有几个参数？ 1个

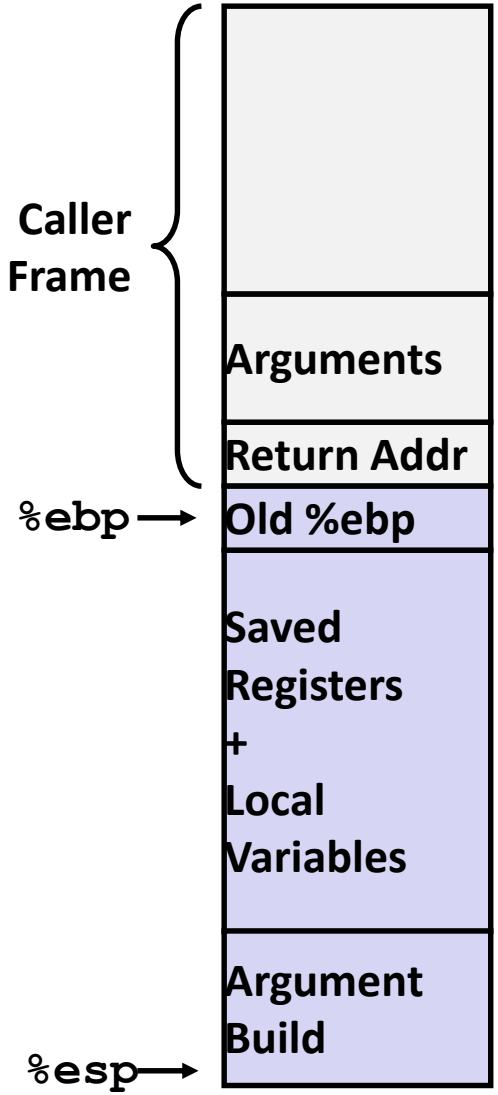
处理结构是怎样的？ 循环结构！

① 处为i=0，② 处为i≠32，③ 处为i++。

入口参数 x 在EBX中，返回参数 result 在EAX中。LEA实现“2*result”，即：将result左移一位；第6和第7条指令则实现“x&0x01”；第8条指令实现“result=(result<<1) | (x & 0x01)”，第9条指令实现“x>>=1”。综上所述，④ 处的C语言语句是“result=(result<<1) | (x & 0x01); x>>=1;”。

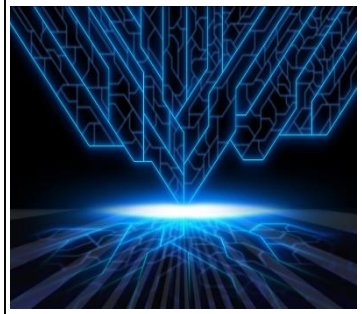
IA 32 过程总结

- 要点
  - 对于过程的调用和返回来说，堆栈是一种非常有效的数据结构
    - If P calls Q, then Q returns before P
- 递归遵循正常的函数调用
  - 能够安全保存变量和被调用程序保护的寄存器到堆栈框架中
  - 函数自变量放到堆栈顶部
  - 返回结果保存在 **%eax**寄存器中
- 指针是有价值的地址
  - 在堆栈上或全局的



主要内容

数组、结构、联合



基本数据类型

- 整数：存储、操作于通用寄存器中
  - 有符号 vs.无符号根据所使用的指令

Intel	ASM	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int
quad word	q	8	[unsigned] long int (x86-64)

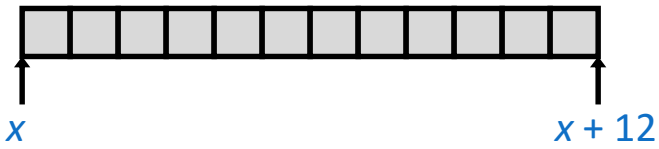
- 浮点数：存储、操作于浮点寄存器中

Intel	ASM	Bytes	C
Single	`s`	4	`float`
Double	`l`	8	`double`
Extended	`t`	10/12/16	`long double`

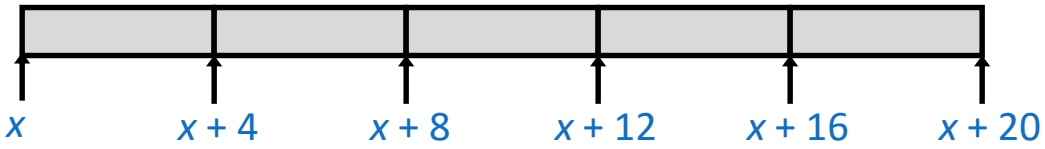
数组分配

- 格式：*T* `A[`*L*`];`
  - 数据类型为T、长度为L的数组
  - 连续分配*L* * `sizeof(`*T*`)` 字节空间

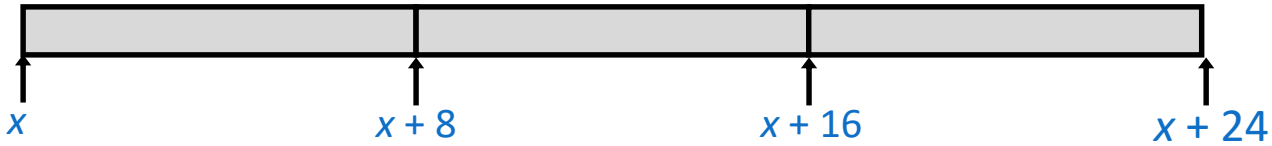
`char string[12];`



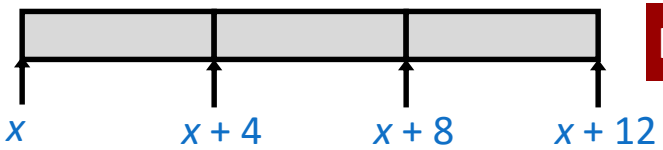
`int val[5];`



`double a[3];`



`char *p[3];`



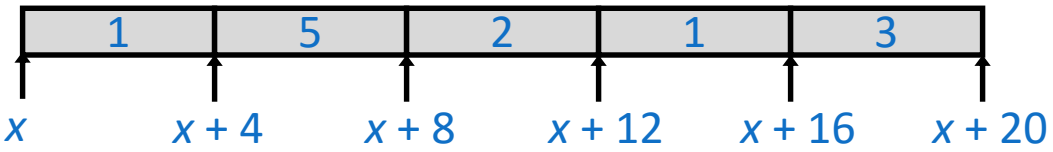
IA32

x86-64

数组访问

- 格式：*T* `A[`*L*`];`
  - 数据类型为T并且长度为L的数组
  - 标识符A能够作为指向数组0号元素的指针：数据类型 T

`int val[5];`

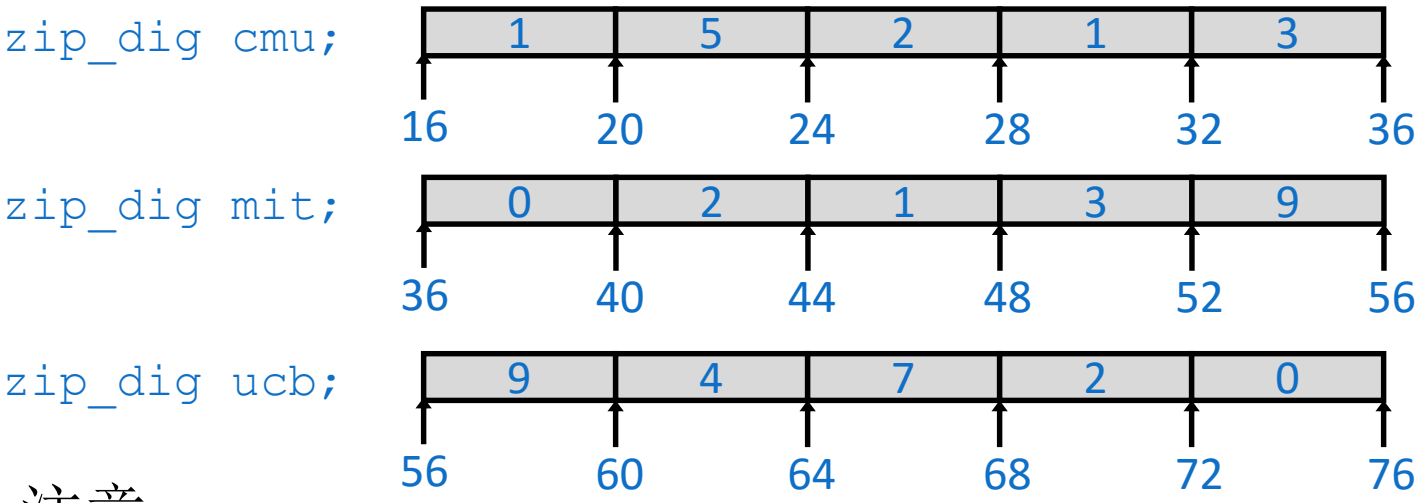


- 引用 类型 值

`val[4]`	`int`	3
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	*x* + 4 *i*

数组例子

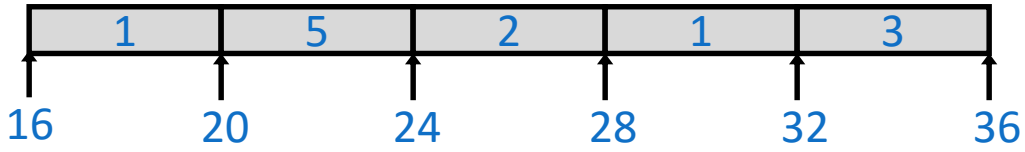
```
#define ZLEN 5
typedef int zip_dig[ZLEN];
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- 注意
  - “zip_dig cmu” 与“int cmu[5]”等价
  - 示例中为3个数组分配了连续的3个20字节块
    - 通常情况下并不能保证数组间的地址连续

数组访问例子

```
zip_dig cmu;
```



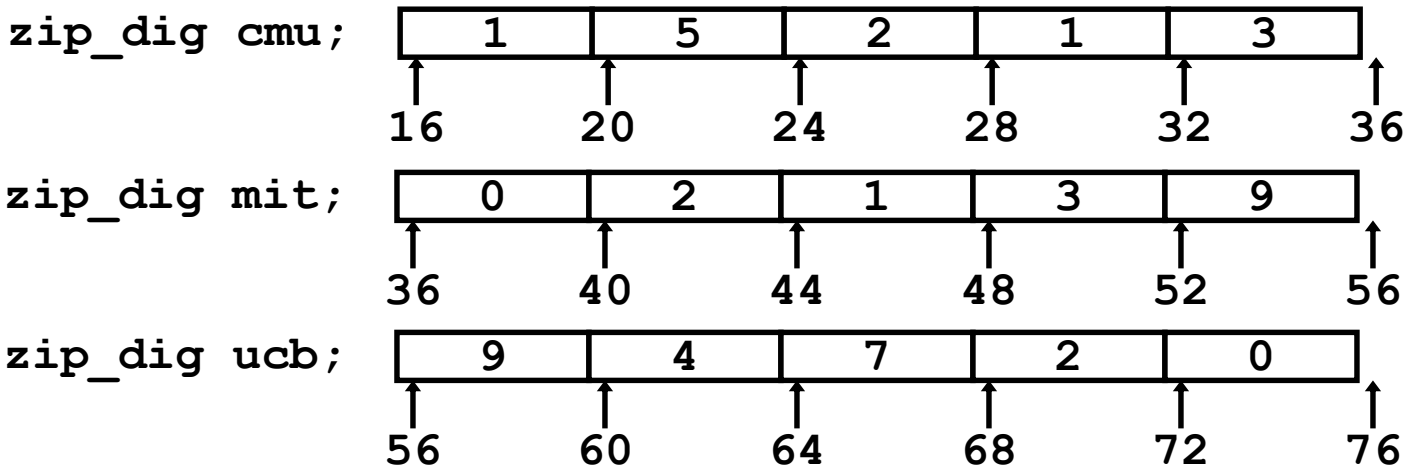

```
int get_digit
  (zip_dig z, int dig)
{
  return z[dig];
}
```

IA32

- 运算
  - 寄存器 %edx 包含数组起始地址
  - 寄存器%eax 包含下标
  - 期望数据在：4*%eax + %edx
  - 使用存储器引用 (%edx,%eax,4)

```
  # %edx = z
  # %eax = dig
  movl (%edx,%eax,4),%eax  # z[dig]
```

引用的例子



- 代码不做任何边界检测
- 引用　　地址　　值　　有保证否？

mit[3]	36 + 4* 3 = 48	3	Yes
mit[5]	36 + 4* 5 = 56	9	No
mit[-1]	36 + 4*-1 = 32	3	No
cmu[15]	16 + 4*15 = 76	??	No

  - 越界的行为依赖于实现
  - 不同数组相对位置关系不能保证

数组循环例子

- 最初代码
  - 我们如何实现它?
  - 能提高它的性能吗?

```
void zincr(zip_dig z) {
  int i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

```
  # edx = z
  movl  $0, %eax            #    %eax = i
.L4:                        # loop:
  addl  $1, (%edx,%eax,4)       #    z[i]++
  addl  $1, %eax            #    i++
  cmpl  $5, %eax            #    i:5
  jne   .L4                 #    if !=, goto
loop
```

数组的分配和访问

- 数组元素在内存的存放和访问
  - 例如，定义一个具有4个元素的静态存储型 short 数据类型数组A，可以写成“static short A[4];”
  - 第 i（0≤i≤3）个元素的地址计算公式为&A[0]+2*i。
  - 假定数组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
```

其中，ECX为变址（索引）寄存器，在循环体中增量 **比例因子是2！**

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

数组元素在内存的存放和访问

- 分配在静态区的数组的初始化和访问

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

buf是在静态区分配的数组，链接后，buf在可执行目标文件的数据段中分配了空间

```
08049908 <buf>:
08049908:    0A 00 00 00 14 00 00 00
```

此时，buf=&buf[0]=0x08049908

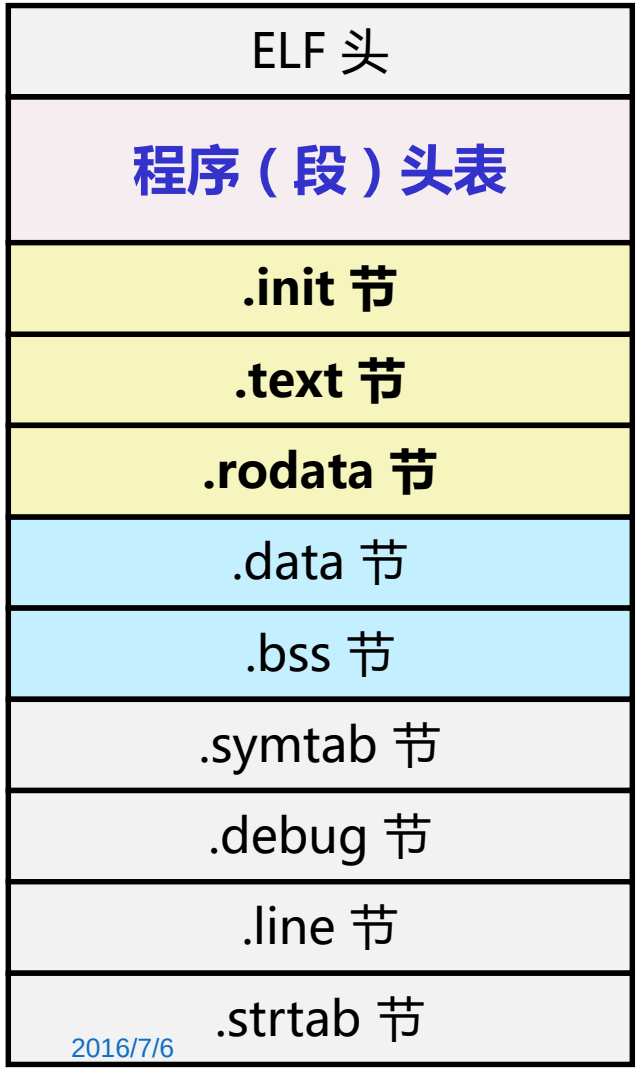
编译器通常将其先存放到寄存器(如EDX)中

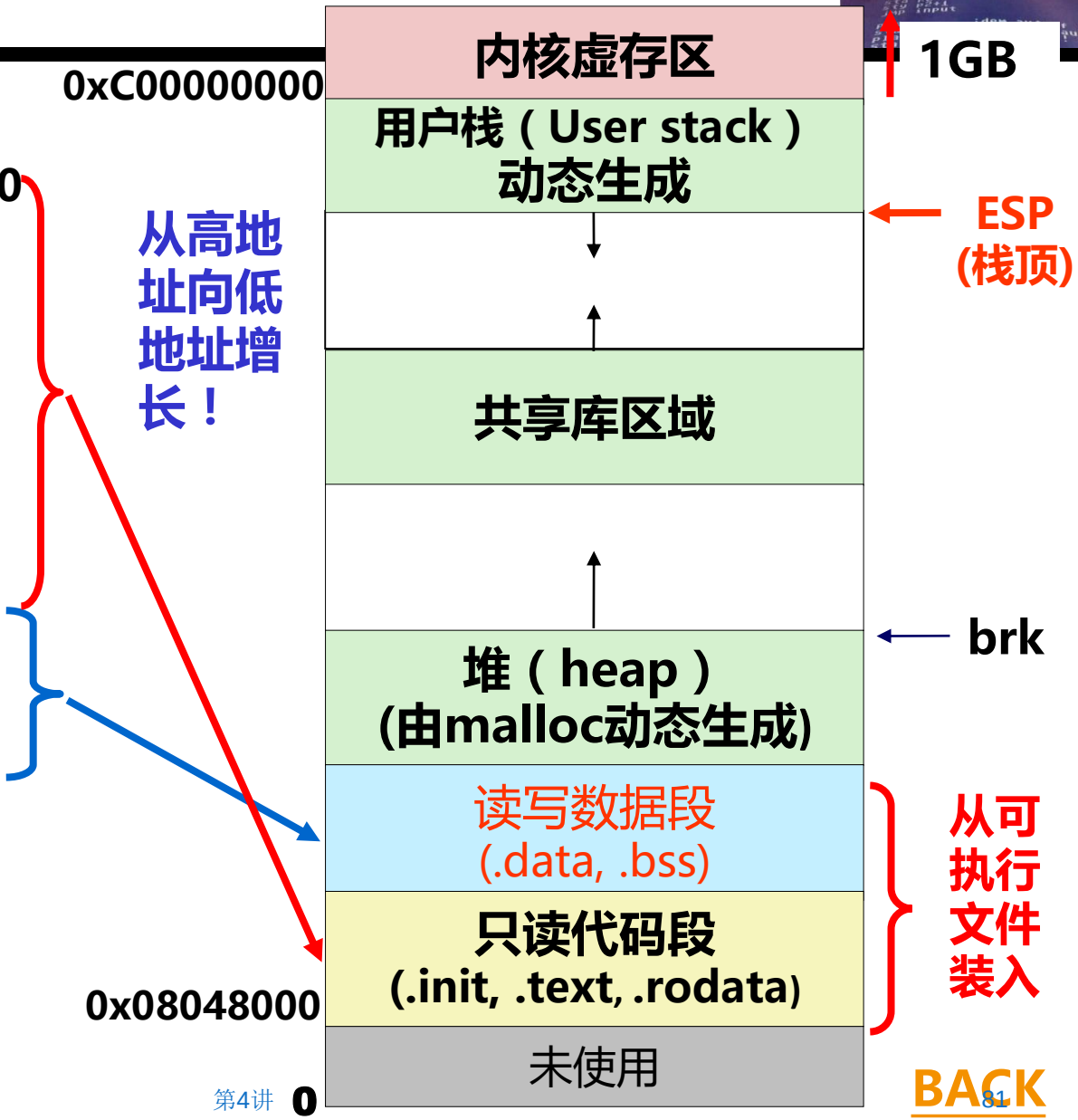
假定 i 被分配在ECX中，sum被分配在EAX中，则“sum+=buf[i];”和i++ 可用什么指令实现？

```
addl buf( , %ecx, 4), %eax 或 addl 0(%edx , %ecx, 4), %eax
addl  $1, %ecx
```

可执行文件的存储器映像

程序(段)头表描述如何映射





BACK

数组元素在内存的存放和访问

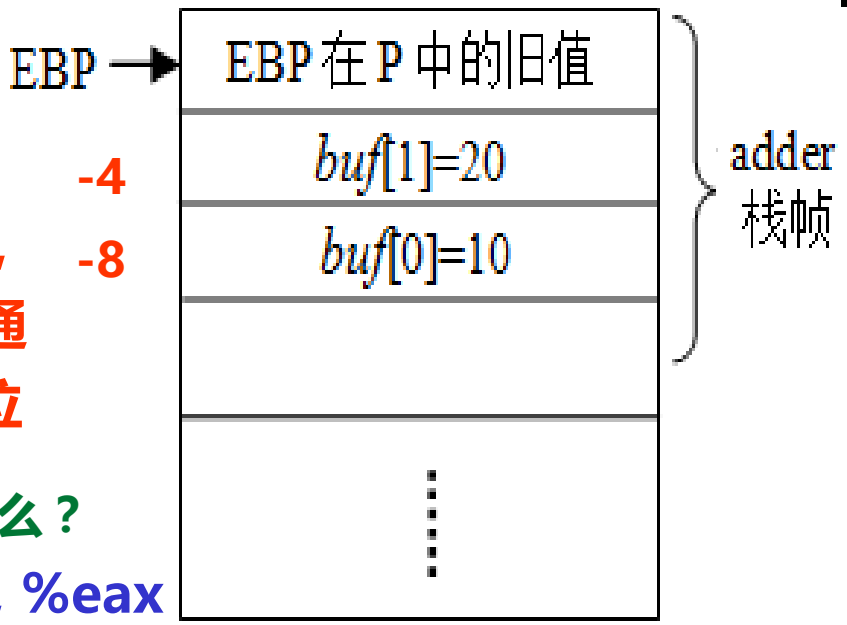
- auto型数组的初始化和访问

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

分配在栈中，故数组首址通过EBP来定位

EDX、ECX各是什么？

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



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
```

数组元素在内存的存放和访问

- 数组与指针

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

✓ 以下两个程序段功能完全相同，都是使ptr指向数组a的第0个元素a[0]

a的值就是其首地址，即 a=&a[0]

因而 a=ptr，从而有

&a[i]=ptr+i=a+i 以及

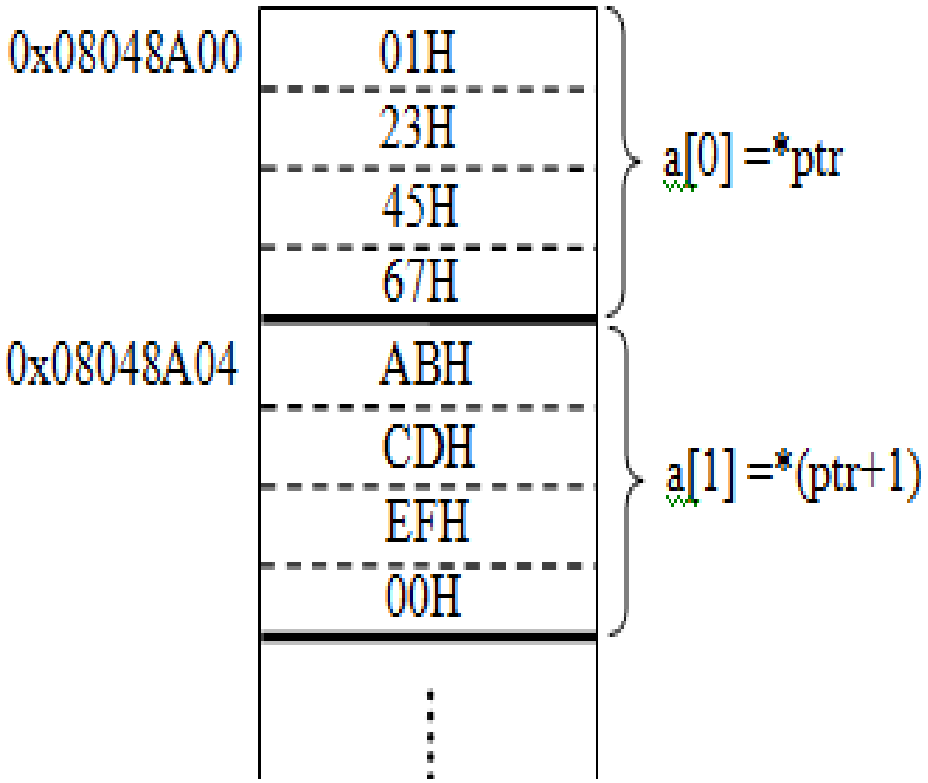
a[i]=ptr[i]=*(ptr+i)=*(a+i)。

（1）int a[10];

int *ptr=&a[0];

（2）int a[10], *ptr;

ptr=&a[0];



小端方式下a[0]=?, a[1]=?

a[0]=0x67452301, a[1]=0x0efcdab

数组首址0x8048A00在ptr中，ptr+i 并不是用0x8048A00加 i 得到，而是等于

0x8048A00+4*i

数组元素在内存的存放和访问

序号	表达式	类型	值的计算方式	汇编代码
1	A	int *		
2	A[0]	int		
3	A[i]	int		
4	&A[3]	int *		
5	&A[i]-A	int		
6	*(A+i)	int		
7	*(&A[0]+i-1)	int		
8	A+i	int *		

问题：

假定数组A的首址SA在ECX中，i在EDX中，表达式结果在EAX中，各表达式的计算方式以及汇编代码各是什么？

2、3、6和7对应汇编指令都需访存，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，因为数组元素的类型为int型，故比例因子为4。

数组元素在内存的存放和访问

假设A首址SA在ECX，i 在EDX，结果在EAX

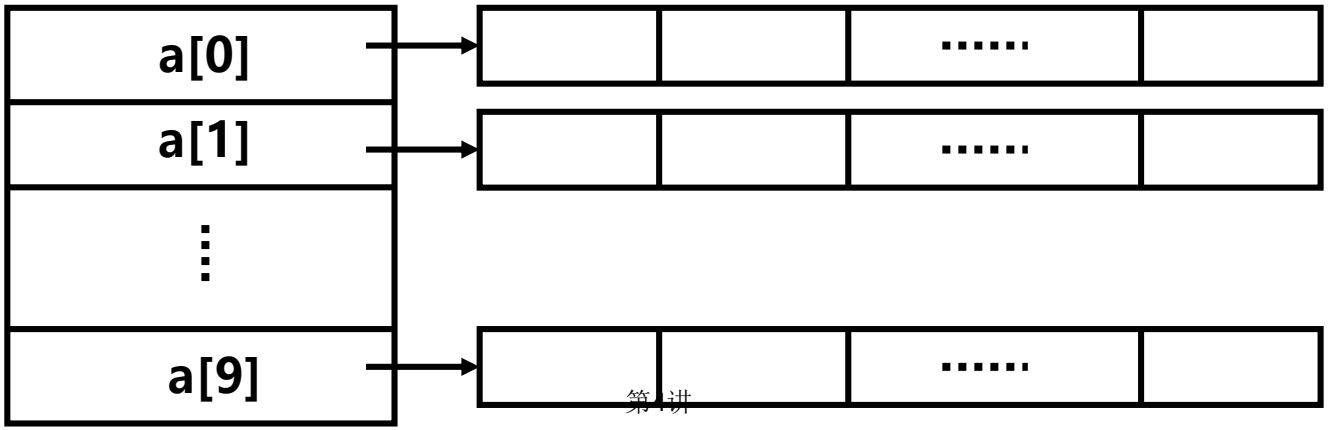
序号	表达式	类型	值的计算方式	汇编代码
1	A	int *	*SA*	leal (%ecx), %eax
2	A[0]	int	M[*SA*]	movl (%ecx), %eax
3	A[i]	int	M[*SA*+4**i*]	movl (%ecx, %edx, 4), %eax
4	&A[3]	int *	*SA*+12	leal 12(%ecx), %eax
5	&A[i]−A	int	(*SA*+4**i*−*SA*)/4=*i*	movl %edx, %eax
6	*(A+i)	int	M[*SA*+4**i*]	movl (%ecx, %edx, 4), %eax
7	*(&A[0]+i−1)	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对应汇编指令都需访存，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，因为数组元素的类型为int型，故比例因子为4 。

数组元素在内存的存放和访问

- 指针数组和多维数组
  - 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。
    - 一个指针数组可以实现一个二维数组。



数组元素在内存的存放和访问

- 指针数组和多维数组

按行优先方式存放数组元素

计算一个两行四列整数矩阵中每一行数据的和。

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

当i=1时，pn[i]=*(pn+i)=M[pn+4*i]=0x8049308

若处理“s[i]+=*pn[i]++;”时 i 在ECX，s[i]在AX，pn[i]在EDX，则对应指令序列可以是什么？

```
movl   pn(,%ecx,4), %edx
addw   (%edx), %ax
addl   $2, pn(, %ecx, 4)
```

pn[i]+”1”→pn[i]

若num=0x8049300,则num、pn和s在存储区中如何存放？

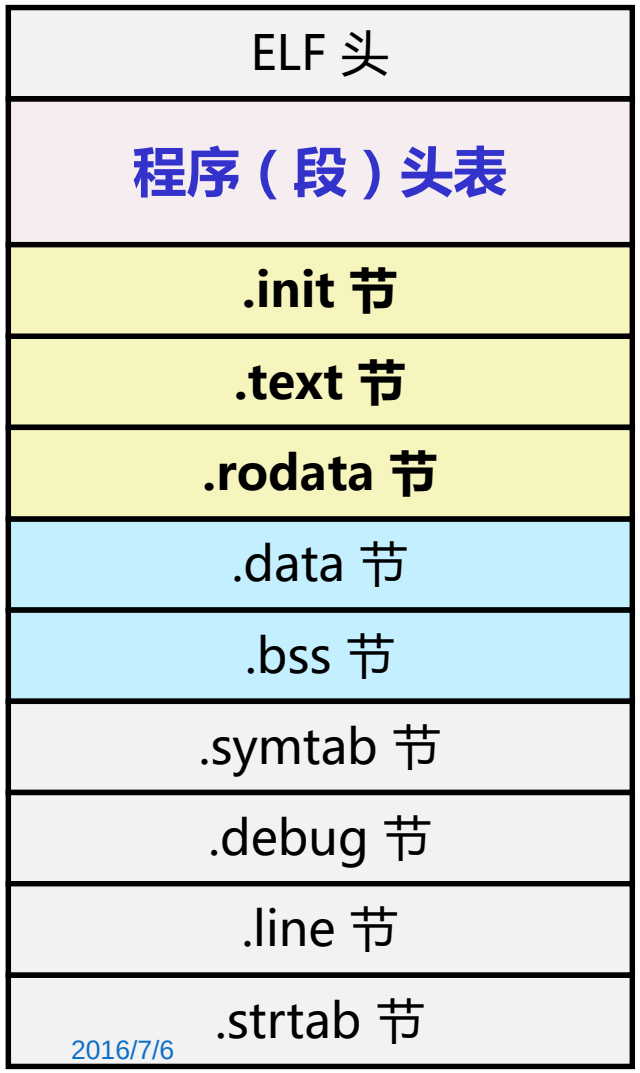
```
08049300 <num>:
08049300 :   02 00 09 00 ff ff 05 00 03 00 08 00 02 00 fa ff
08049310 <pn>:
08049310 :   00 93 04 08 08 93 04 08
08049318<s>:
08049318 :   00 00 00 00
```

num=num[0]=&num[0][0]=0x8049300

pn=&pn[0]=0x8049310
pn[0]=num[0]=0x8048300
pn[1]=num[1]=0x8048308

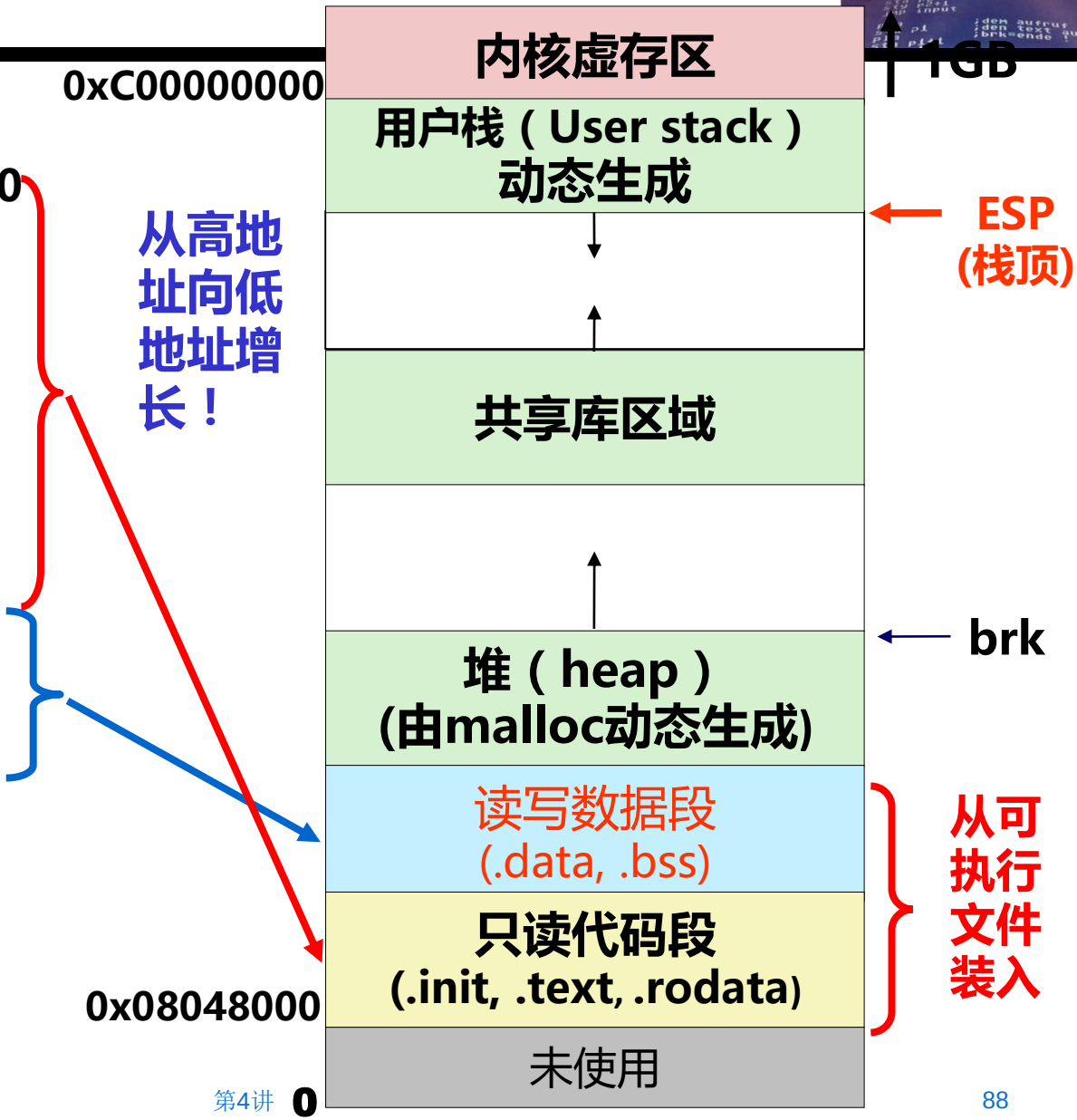
可执行文件的存储器映像

程序(段)头表描述如何映射



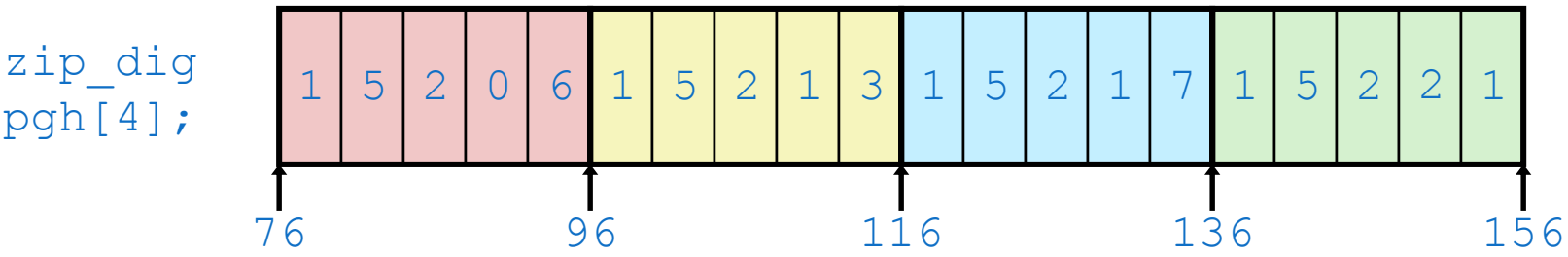
0

从高地址向低地址增长！



嵌套数组例子

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



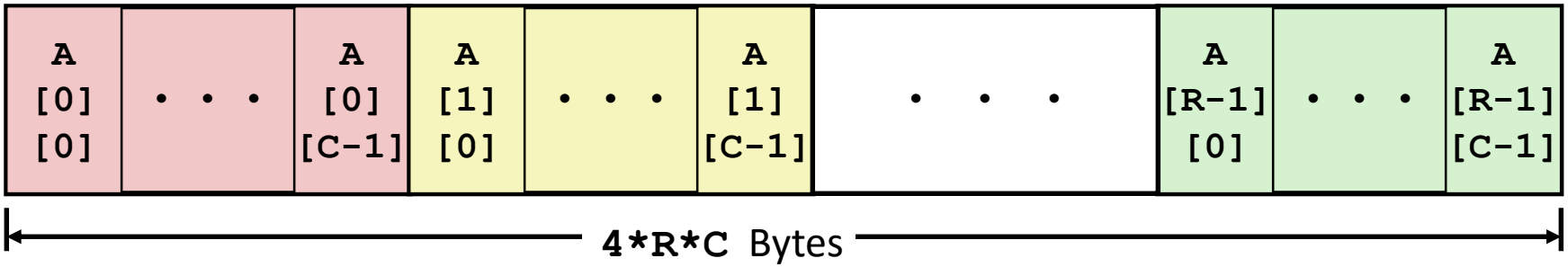
- 声明“zip_dig pgh[4]”⇔“int pgh[4][5]”
  - pgh 变量表示4个元素的数组，分配连续空间
  - 每个元素是有5个int类型的数组，分配连续空间
- 保证“行优先”顺序排列元素

多维数组

- 声明：*T* A[*R*][*C*];
  - 数据类型为T的数组
  - *R* 行, *C* 列
  - 类型T的元素需要K字节
- 数组大小
  - *R* * *C* * *K* bytes
- 排列：行优先顺序

```
A[0][0]    • • •   A[0][C-1]
   •                    •
   •                    •
   •                    •
A[R-1][0]  • • •  A[R-1][C-1]
```

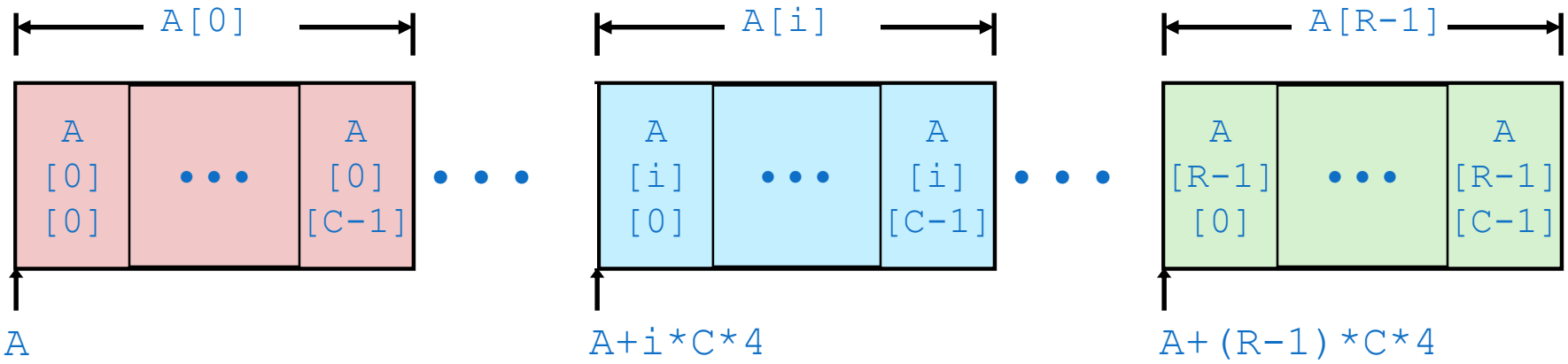
int A[R][C];



嵌套数组的行访问

- 行向量
  - A[i]是含有C个元素的数组
  - 每个元素类型为T
  - 起始地址为A + $i*C*K$

int A[R][C];



嵌套数组行访问代码

```
int *get_pgh_zip(int
index)
{
  return pgh[index];
}
```

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

```
# %eax = index
leal (%eax,%eax,4),%eax     # 5 * index
leal pgh(,%eax,4),%eax      # pgh + (20 * index)
```

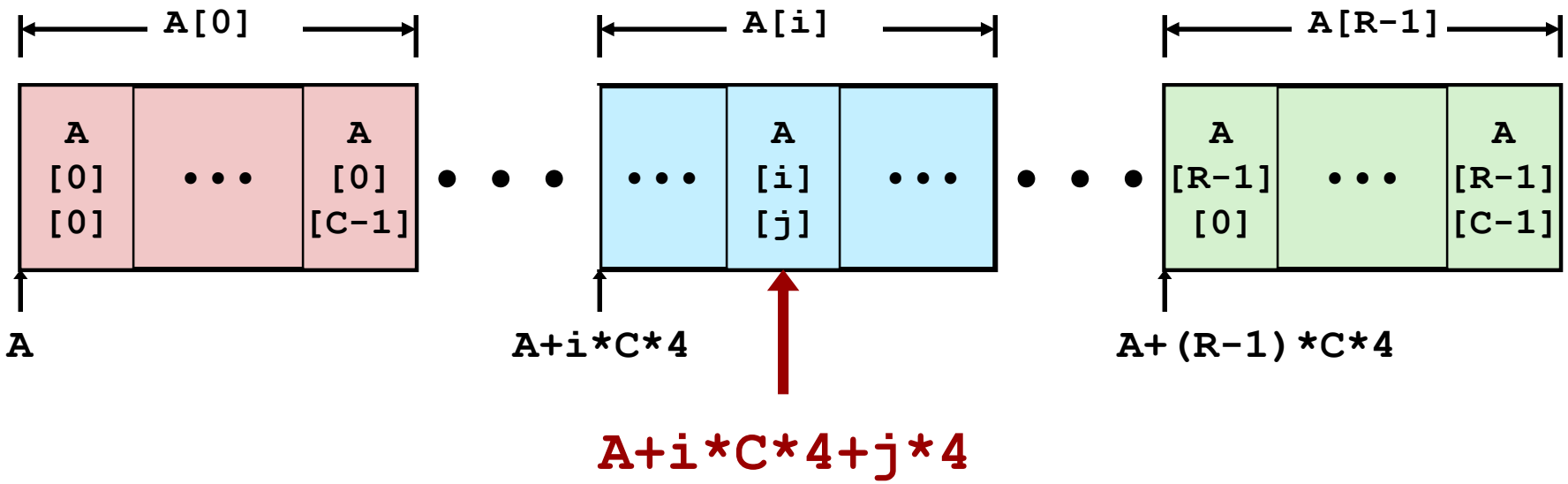
- 行向量
  - pgh[index]是5个int型元素的数组
  - 起始地址为pgh+20*index
- IA32代码
  - 运算并返回地址
  - 计算pgh + 4*(index+4*index)

嵌套数组元素的访问

- 数组元素
  - A[i][j] 是一个T类型的元素
  - 地址为A + (*i* * *C* + *j*) * *K*

```
A
[i]
[j]
```

```
int A[R][C];
```



嵌套数组元素访问代码

```
int get_pgh_digit
   (int index, int dig)
{
  return pgh[index][dig];
}
```

```
movl   8(%ebp), %eax          # index
leal   (%eax,%eax,4), %eax    # 5*index
addl   12(%ebp), %eax         # 5*index+dig
movl   pgh(,%eax,4), %eax     # offset 4*(5*index+dig)
```

- 数组元素 pgh[index][dig]是一个int型
  - 地址: pgh + 20*index + 4*dig

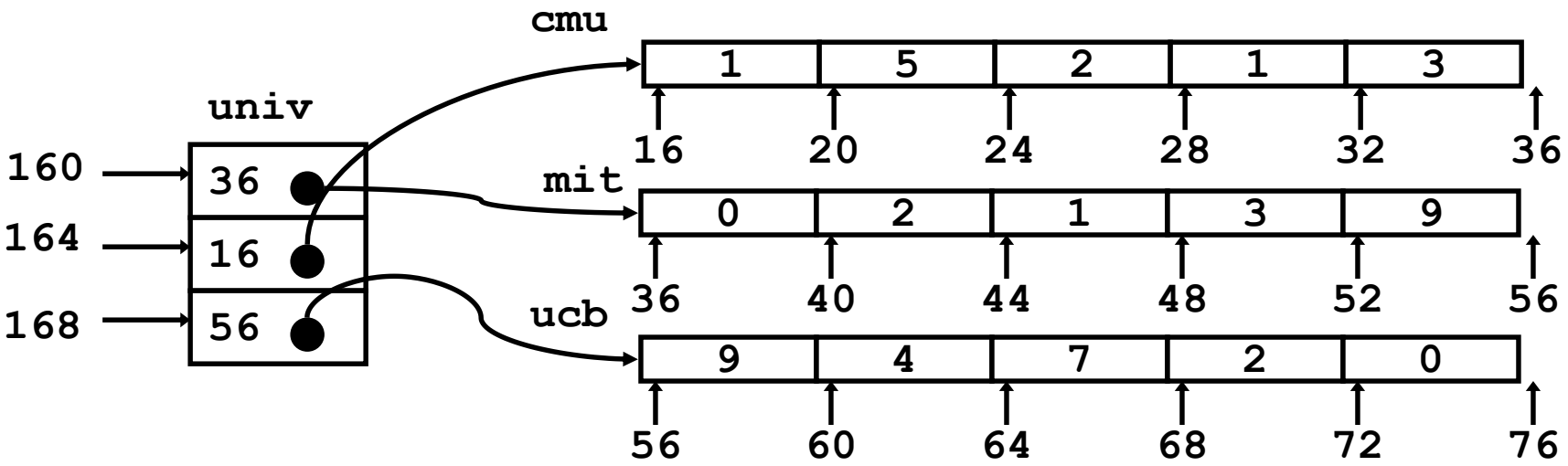
    = pgh + 4*(5*index + dig)
- IA32代码
  - 运算地址：**pgh + 4*((index+4*index)+dig)**
  - movl 执行存储器引用

多层数组Multi-Level Array

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, cmu, ucb};
```

- 变量univ表示3个元素的数组
- 每个元素是一个指针
  - 4 字节
- 每个指针指向一个int类型数组



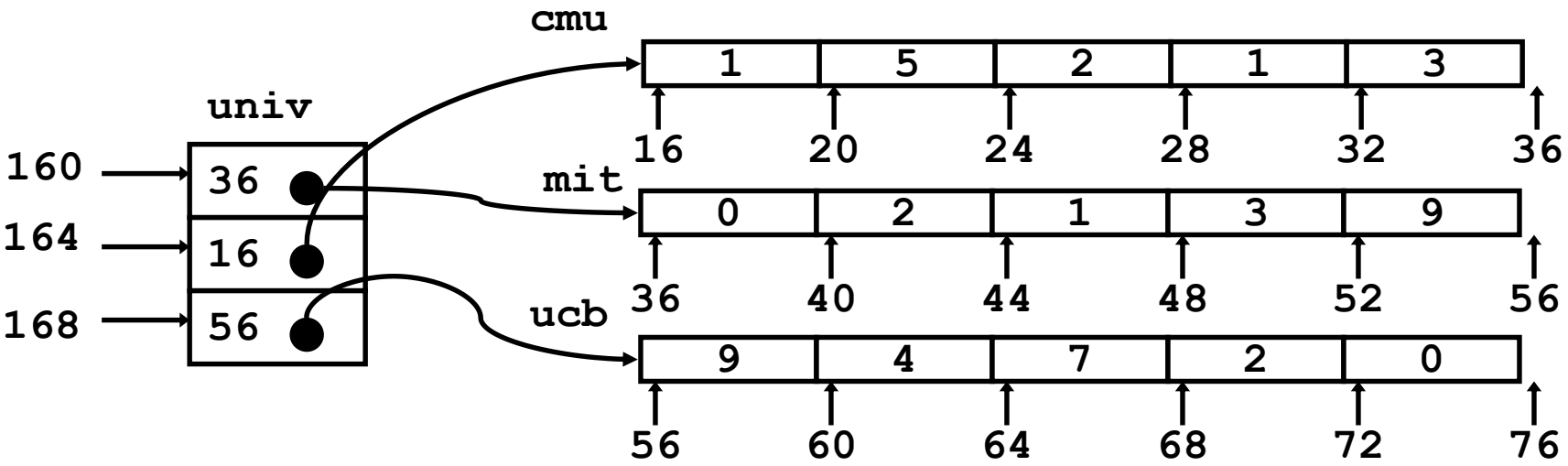
多层数组中元素访问

```
int get_univ_digit(int index, int
dig)
{
  return univ[index][dig];
}
```

```
movl    8(%ebp), %eax          # index
movl    univ(,%eax,4), %edx    # p = univ[index]
movl    12(%ebp), %eax         # dig
movl    (%edx,%eax,4), %eax    # p[dig]
```

- 运算
  - 元素访问Mem[Mem[univ+4*index]+4*dig]
  - 必须做两次存储读操作
    - 首先获得指向行数组的指针
    - 然后在数组内访问元素

存储引用的例子



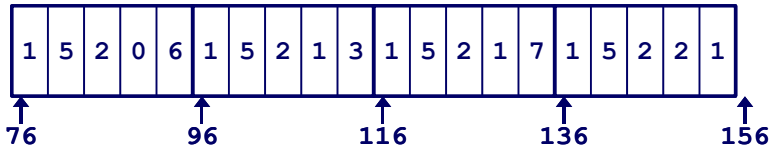
引用	地址	值	Guaranteed?
univ[2][3]	56+4*3 = 68	2	Yes
univ[1][5]	16+4*5 = 36	0	No
univ[2][-1]	56+4*-1 = 52	9	No
univ[3][-1]	??	??	No
univ[1][12]	16+4*12 = 64	7	No

- 代码从不做任何边界检测
- 不同数组中元素顺序不能保证

数组元素访问

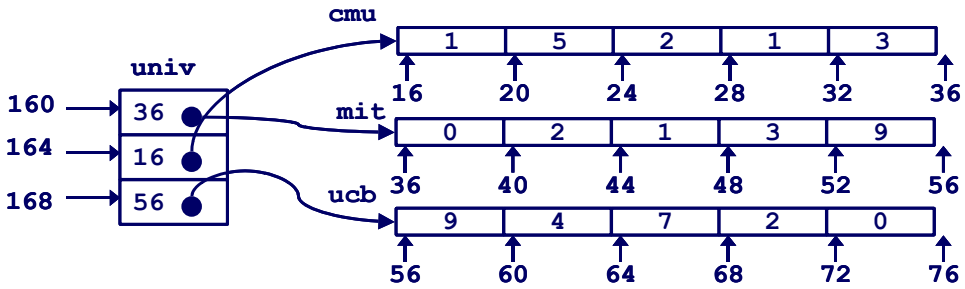
- 嵌套数组

```
int get_pgh_digit
   (int index, int dig)
{
   return pgh[index][dig];
}
```



- 多级数组

```
int get_univ_digit
   (int index, int dig)
{
   return univ[index][dig];
}
```



语法是相同的，地址运算却是完全不同的!

Mem[pgh+20*index+4*dig]

Mem[Mem[univ+4*index]+4*dig]

N*N矩阵

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele
  (fix_matrix a, int i, int j){
  return a[i][j];
}
```

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element a[i][j] */
int vec_ele
 (int n, int *a, int i, int j){
  return a[IDX(n,i,j)];
}
```

```
/* Get element a[i][j] */
int var_ele
 (int n, int a[n][n], int i, int j){
  return a[i][j];
}
```

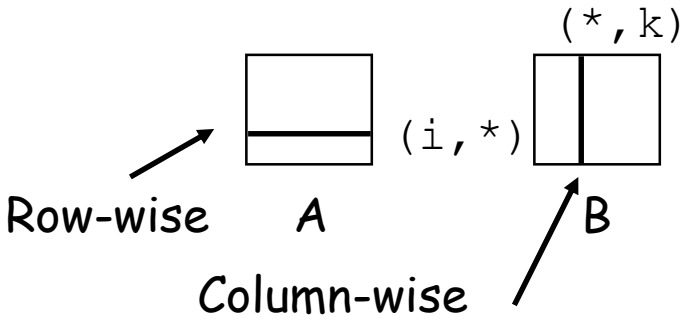
- 固定维数
  - 在编译的时N数值确定
- 可变维数,直接寻址
  - 传统方法是利用动态数组
- 可变维数,间接寻址
  - gcc支持

16*16矩阵

```
/* Get element a[i][j] */
int fix_ele(fix_matrix a, int i, int j) {
  return a[i][j];
}
```

```
movl    12(%ebp), %edx   # i
sall    $6, %edx         # i*64
movl    16(%ebp), %eax   # j
sall    $2, %eax         # j*4
addl    8(%ebp), %eax    # a + j*4
movl    (%eax,%edx), %eax     # *(a + j*4 + i*64)
```

- **数组元素**
  - 地址：**A** + *i* * (*C* * *K*) + *j* * *K*
  - C = 16, K = 4



n X n Matrix Access

```
/* Get element a[i][j] */
int var_ele(int n, int a[n][n], int i, int j) {
  return a[i][j];
}
```

```
  movl   8(%ebp), %eax      # n
  sall   $2, %eax           # n*4
  movl   %eax, %edx         # n*4
  imull  16(%ebp), %edx     # i*n*4
  movl   20(%ebp), %eax     # j
  sall   $2, %eax           # j*4
  addl   12(%ebp), %eax     # a + j*4
  movl   (%eax,%edx), %eax  # *(a + j*4 + i*n*4)
```

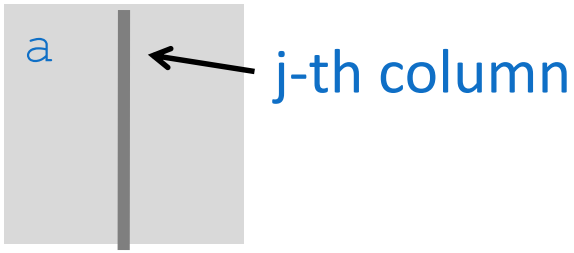
- **Array Elements**
  - Address **A** + $i * (C * K) + j * K$
  - C = n, K = 4

固定维数数组优化

```
#define N 16
typedef int fix_matrix[N][N];
```

```
/* Retrieve column j from array */
void fix_column
  (fix_matrix a, int j, int *dest)
{
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

- 所有元素排列在1行
- 优化：从1行中搜索元素



固定维数数组优化

- 优化
  - 计算 ajp = &a[i][j]
    - 初始化= a + 4*j
    - 增量 4*N

Register	Value
%ecx	ajp
%ebx	dest
%edx	i

```
/* Retrieve column j from array */
void fix_column
  (fix_matrix a, int j, int *dest)
{
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

在下标运算中避免了乘法

```
.L8:                              # loop:
   movl  (%ecx), %eax             #   Read *ajp
   movl  %eax, (%ebx,%edx,4)      #   Save in dest[i]
   addl  $1, %edx                 #   i++
   addl  $64, %ecx                #   ajp += 4*N
   cmpl  $16, %edx                #   i:N
   jne   .L8                      #   if !=, goto loop
```

动态嵌套数组

- 优势
  - 能够创建任意大小的矩阵
- 编程
  - 必须显示计算下标
- 性能
  - 访问单一元素代价高
  - 必须做乘法

```
int * new_var_matrix(int n)
{
  return (int *)
    calloc(sizeof(int), n*n);
}
```

```
int var_ele
  (int *a, int i,
   int j, int n)
{
  return a[i*n+j];
}
```

```
movl 12(%ebp),%eax        # i
movl 8(%ebp),%edx         # a
imull 20(%ebp),%eax       # n*i
addl 16(%ebp),%eax        # n*i+j
movl (%edx,%eax,4),%eax   # Mem[a+4*(i*n+j)]
```

动态数组优化

- 计算 ajp = &a[i][j]
  - Initially = a + 4*j
  - Increment by 4*n

Register	Value
%ecx	ajp
%edi	dest
%edx	i
%ebx	4*n
%esi	n

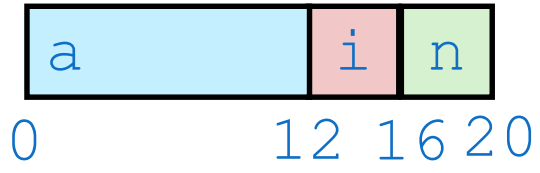
```
/* Retrieve column j from array */
void var_column
  (int n, int a[n][n],
   int j, int *dest)
{
  int i;
  for (i = 0; i < n; i++)
    dest[i] = a[i][j];
}
```

```
.L18:                           # loop:
   movl  (%ecx), %eax           #   Read *ajp
   movl  %eax, (%edi,%edx,4)    #   Save in dest[i]
   addl  $1, %edx               #   i++
   addl  $ebx, %ecx             #   ajp += 4*n
   cmpl  $edx, %esi             #   n:i
   jg    .L18                   #   if >, goto loop
```

结构

```
struct rec {
   int a[3];
   int i;
   struct rec *n;
};
```

存储布局



- 概念
  - 连续分配的存储空间
  - 通过名字引用结构成员
  - 成员可能是不同的类型
- 访问结构元素
  - 指针指向结构的第1个字节
  - 通过地址偏移来访问不同元素

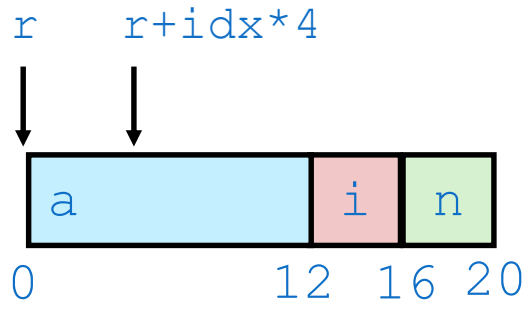
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

IA32汇编代码

```
# %edx = val
# %eax = r
movl %edx, 12(%eax)  # Mem[r+12] = val
```

生成指向结构成员的指针

```
struct rec {
  int a[3];
  int i;
  struct rec *n;
};
```



- 生成指向数组元素的指针
  - 在编译前确定每个结构成员的偏移量
  - 参数
    - Mem[%ebp+8]: **r**
    - Mem[%ebp+12]: **idx**

```
int *get_ap
  (struct rec *r, int idx)
{
  return &r->a[idx];
}
```

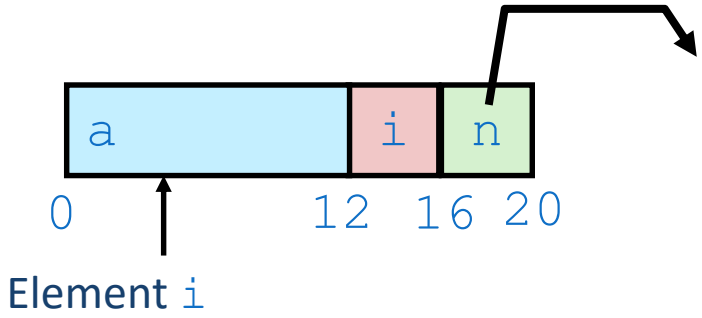
```
movl   12(%ebp), %eax   # Get idx
sall   $2, %eax         # idx*4
addl   8(%ebp), %eax    # r+idx*4
```

结构引用（续）

- C 代码

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->n;
  }
}
```

```
struct rec {
  int a[3];
  int i;
  struct rec *n;
};
```




```
.L17:                           # loop:
  movl   12(%edx), %eax         # r->i
  movl   %ecx, (%edx,%eax,4)    # r->a[i] = val
  movl   16(%edx), %edx         # r = r->n
  testl  %edx, %edx             # Test r
  jne    .L17                   # If != 0 goto loop
```

结构体数据的分配和访问

- 结构体成员在内存的存放和访问
  - 分配在栈中的auto结构型变量的首地址由EBP或ESP来定位
  - 分配在静态区的结构型变量首地址是一个确定的静态区地址
  - 结构型变量 x 各成员首址可用“基址加偏移量”的寻址方式

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

若变量x分配在地址0x8049200开始的区域，那么
x=&(x.id)=0x8049200（若x在EDX中）
&(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

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

x初始化后，在地址0x8049208到0x804920D处是字符串“ZhangS”，0x804920E处是字符‘\0’，从0x804920F到0x8049213处都是空字符。

“unsigned xpost=x.post;”对应汇编指令为“movl 20(%edx), %eax”

结构体数据的分配和访问

- 结构体数据作为入口参数

按地址调用 stu_phone1(&x)

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

按值调用 stu_phone1(x)

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

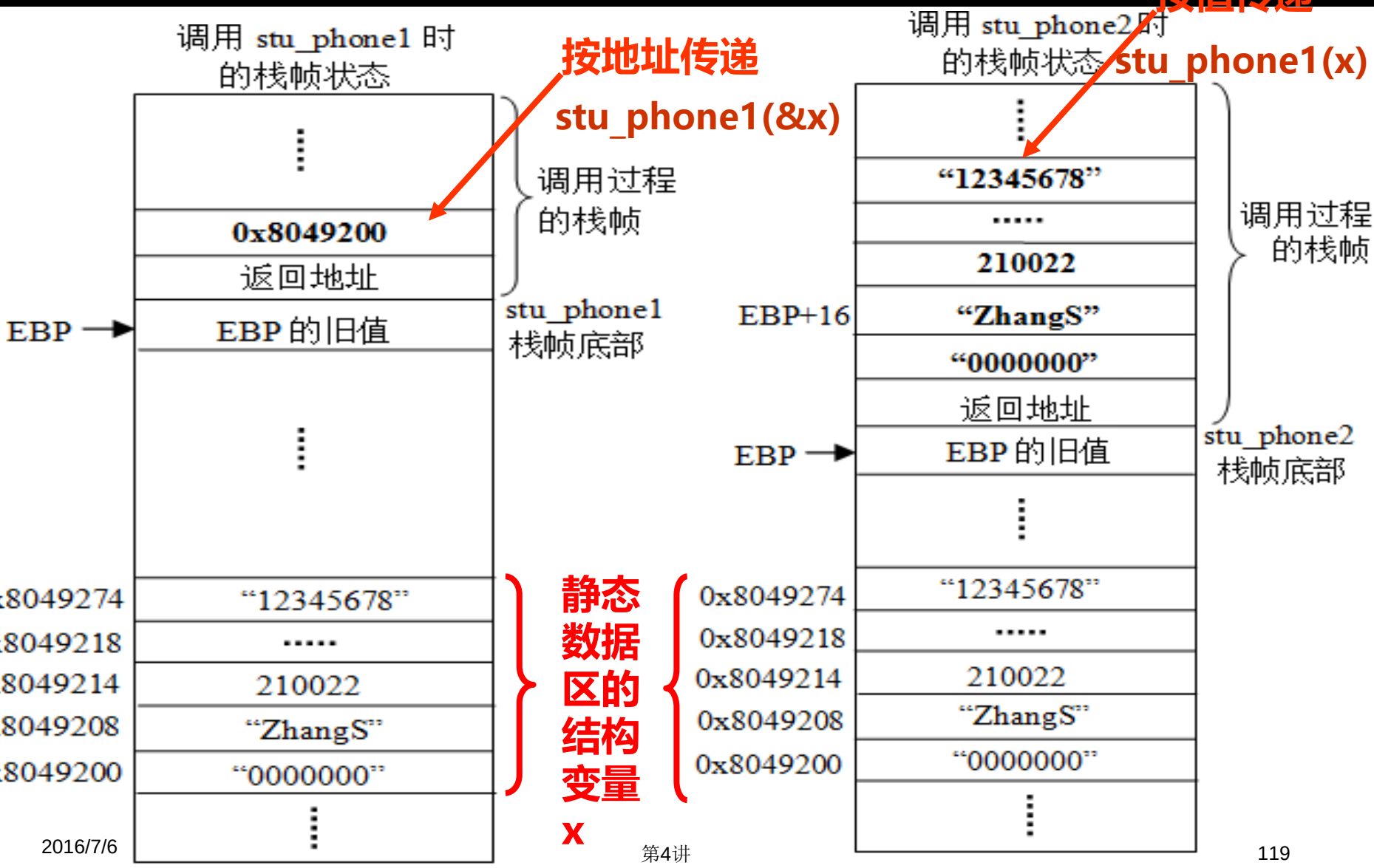
- 当结构体变量需要作为一个函数的形参时，形参和调用函数中的实参应具有相同结构

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

- 若采用按值传递，则结构成员都要复制到栈中参数区，这既增加时间开销又增加空间开销，且更新后的数据无法在调用过程使用(如前面的swap(a,b)例子)
- 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

结构体数据的分配和访问

- 结构体数据作为入口参数（若对应实参是x）



结构体数据的分配和访问

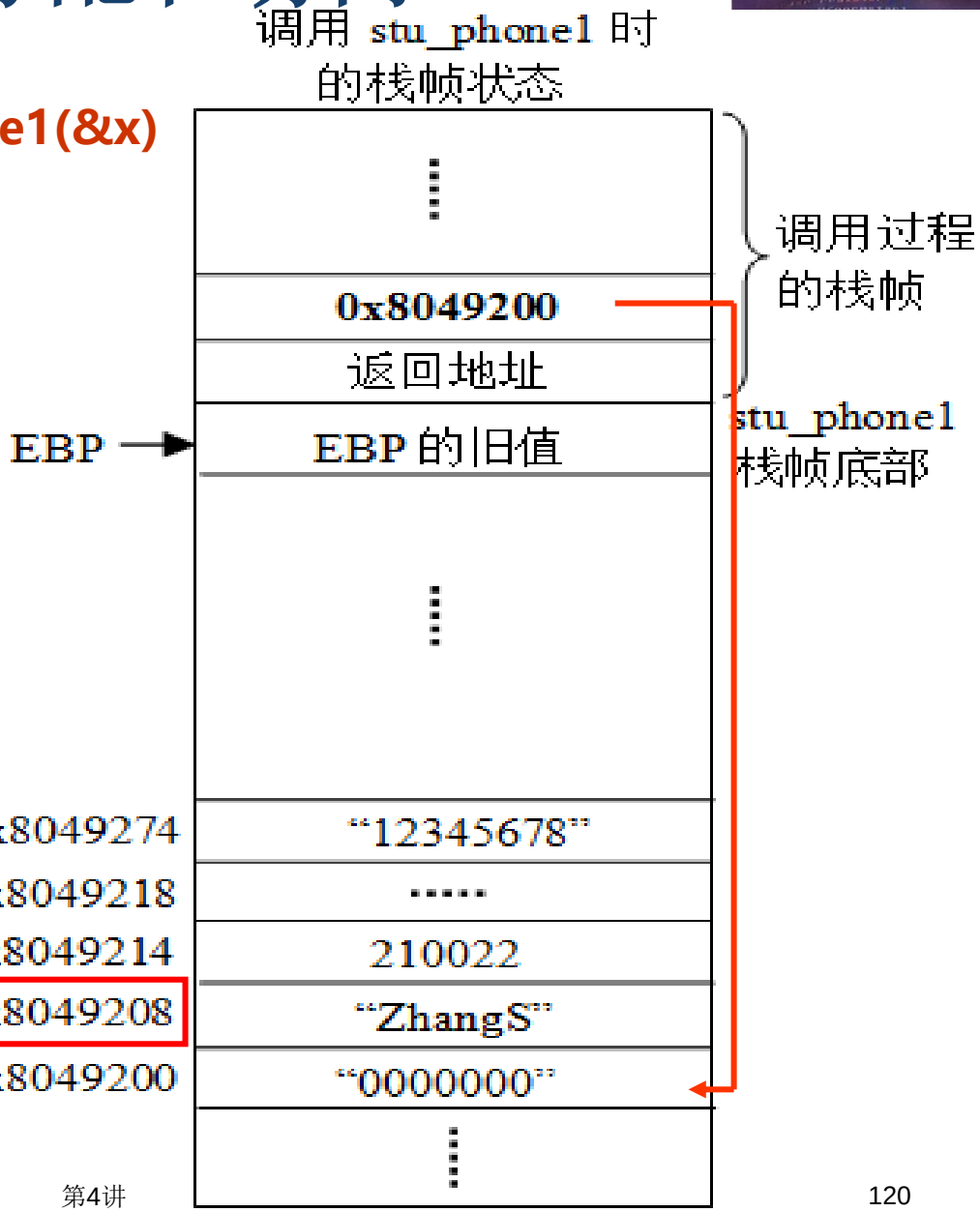
- 按地址传递参数 **stu_phone1(&x)**

(*stu_info).name可写成stu_info->name，执行以下两条指令后:

movl 8(%ebp), edx

leal 8(%edx), eax

EAX中存放的是字符串“ZhangS”在静态存储区内的首地址0x8049208



结构体数据的分配和访问

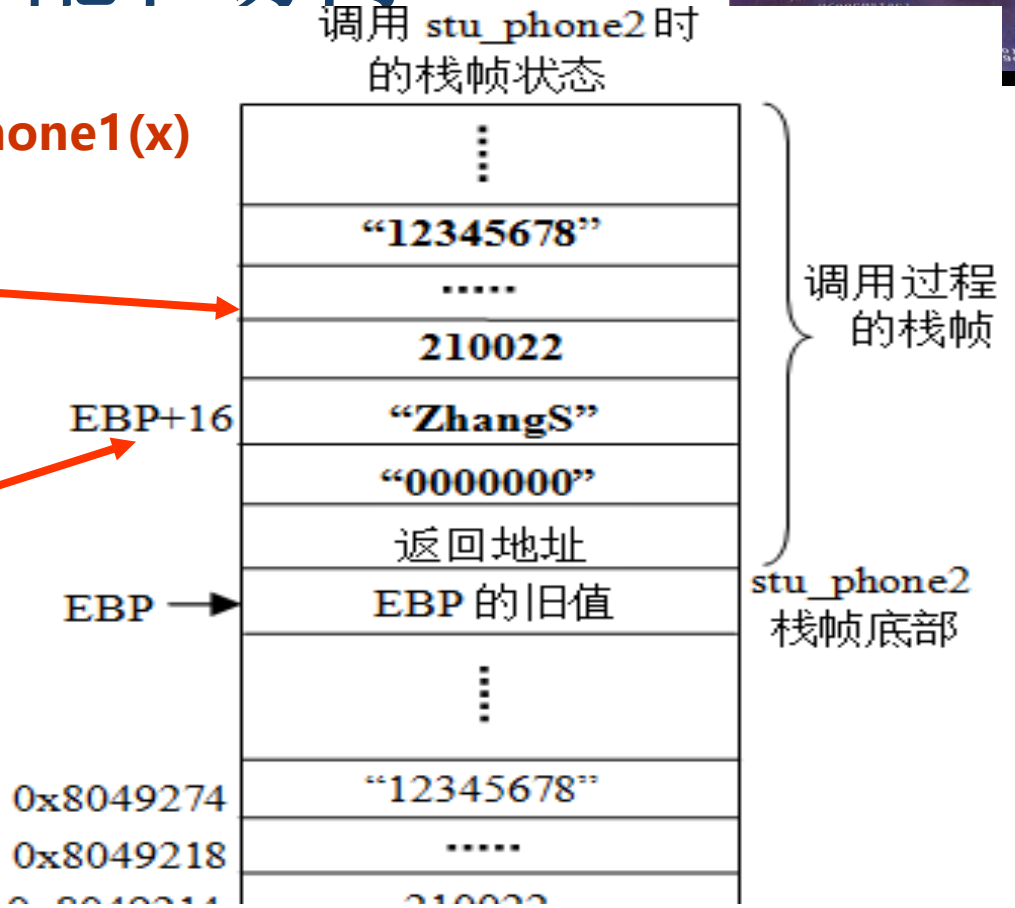
- 按值传递参数

x所有成员值作为实参存到参数区。 stu_info.name送EAX的指令序列为：

leal 8(%ebp), edx

leal 8(%edx), eax

EAX中存放的是“ZhangS”的栈内参数区首址。

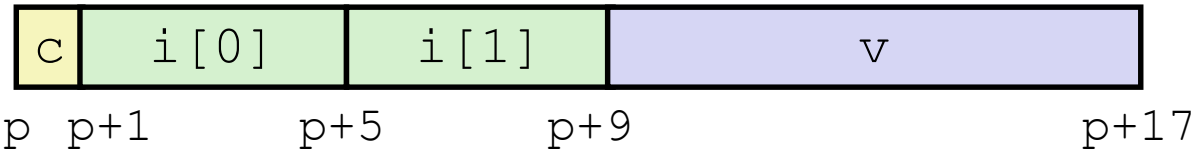


- **stu_phone1和stu_phone2功能相同，但后者开销大，因为它需对结构体成员整体从静态区复制到栈中，需要很多条mov或其他指令，从而执行时间更长，并占更多栈空间和代码空间**

特别是，按值传递时，无法获得更新后的结果

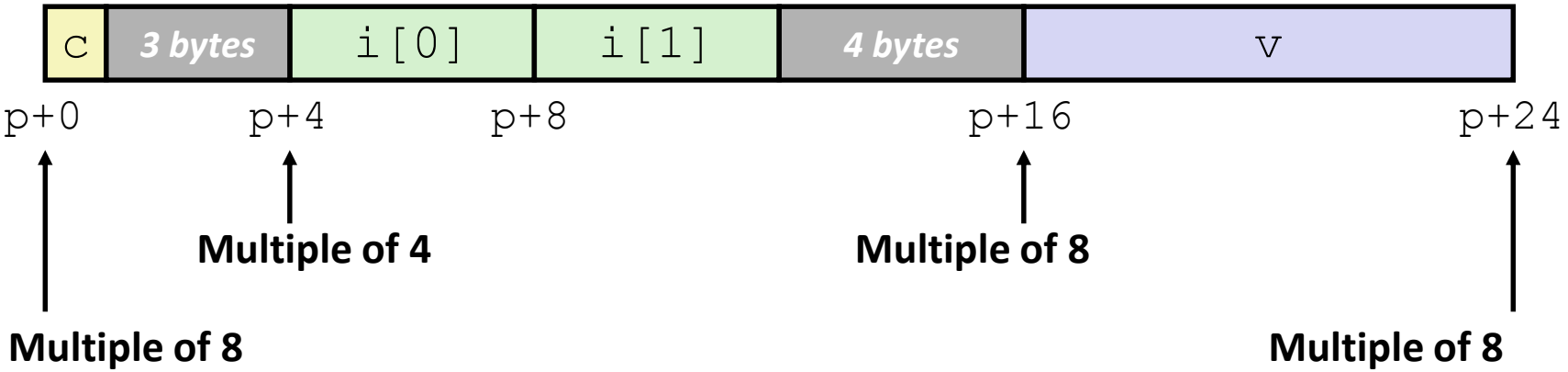
结构与数据对齐

- 不对齐数据分配




```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- 对齐数据分配
  - 原始数据类型需要***K*** 字节
  - 地址必须分配 ***K*** 的倍数开始



数据对齐

- 数据对齐：目的提升系统效率
  - 基本数据类型需要K字节，则数据对应的地址必须是从K的倍数开始
- 对齐数据的目的
  - 存储器访问都是通过对齐的双字或者四字
    - 如果数据跨越了四字边界存取效率低下
    - 虚存中数据跨越两页访问情况会很复杂
- 编译器
  - 在结构中插入一些空隙来确保正确的域对齐
- IA32 Linux 、X86-64 Linux与 Windows处理不同!

IA-32中数据对齐

- 1个字节的数据类型 (如char)：在初始地址上无限制
- 2个字节的数据类型 (如 short):地址的末位必须是0_2，即是2的倍数。
- 4个字节的数据类型 (如 int, float, char *, etc.)
  - 地址的末2位必须是00_2，即是4的倍数。
- 8个字节的数据类型 (如 double)
  - Windows要求：**最低3位地址必须是000_2**，即是8的倍数
  - Linux要求:**最低2位地址必须是00_2，与4字节同样处理**
- 12个字节的数据类型 (long double)
  - Windows,Linux:**最低2位地址必须是00_2，与4字节基本数据类型同样处理**

X86-64中数据对齐

- 1 byte: `char`, …
  - 在地址上无限制
- 2 bytes: `short`, …
  - 最低1位地址必须是0_2
- 4 bytes : `int`, `float`, `char *`, …
  - 最低2位地址必须是00_2
- 8 bytes: `double`,`char*`, …
  - Windows,Linux:
    - **最低3位地址必须是000_2**
- 16 bytes :`long double`, …
  - Linux:
    - **最低3位地址必须是000_2**
    - **i.e.,与8字节基本数据类型同样处理**

数据的对齐

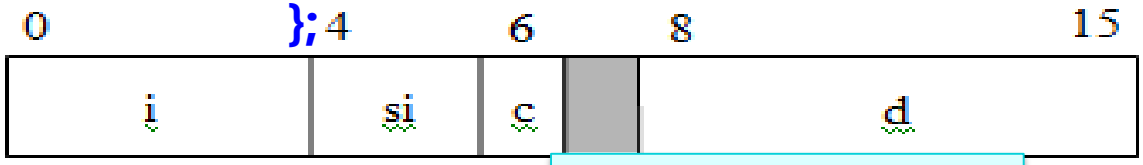
- CPU访问主存时只能一次读取或写入若干特定位。例如，若每次最多读写64位，则第0字节到第7字节可同时读写，第8字节到第15字节可同时读写，……，以此类推。
- 按边界对齐，可使读写数据位于8i~8i+7(i=0,1,2,…) 单元 。
- 最简单的对齐策略是，按其数据长度进行对齐，例如，int型地址是4的倍数，short型地址是2的倍数，double和long long型的是8的倍数，float型的是4的倍数，char不对齐。Windows采用该策略。Linux策略更宽松：short是2的倍数，其他如int、double和指针等都是4的倍数。

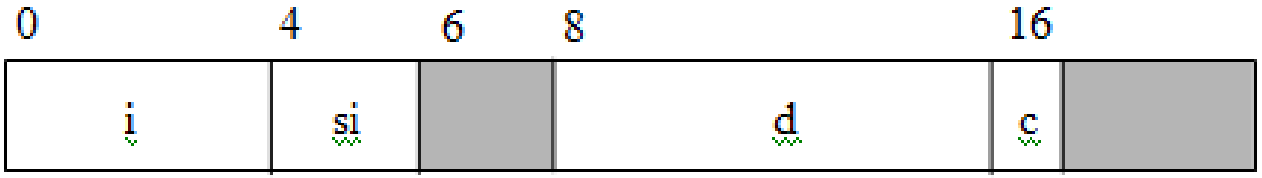
```
struct SDT {
    int     i;
    short   si;
    double  d;
    char    c;
} sa[10];
```

结构数组变量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

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

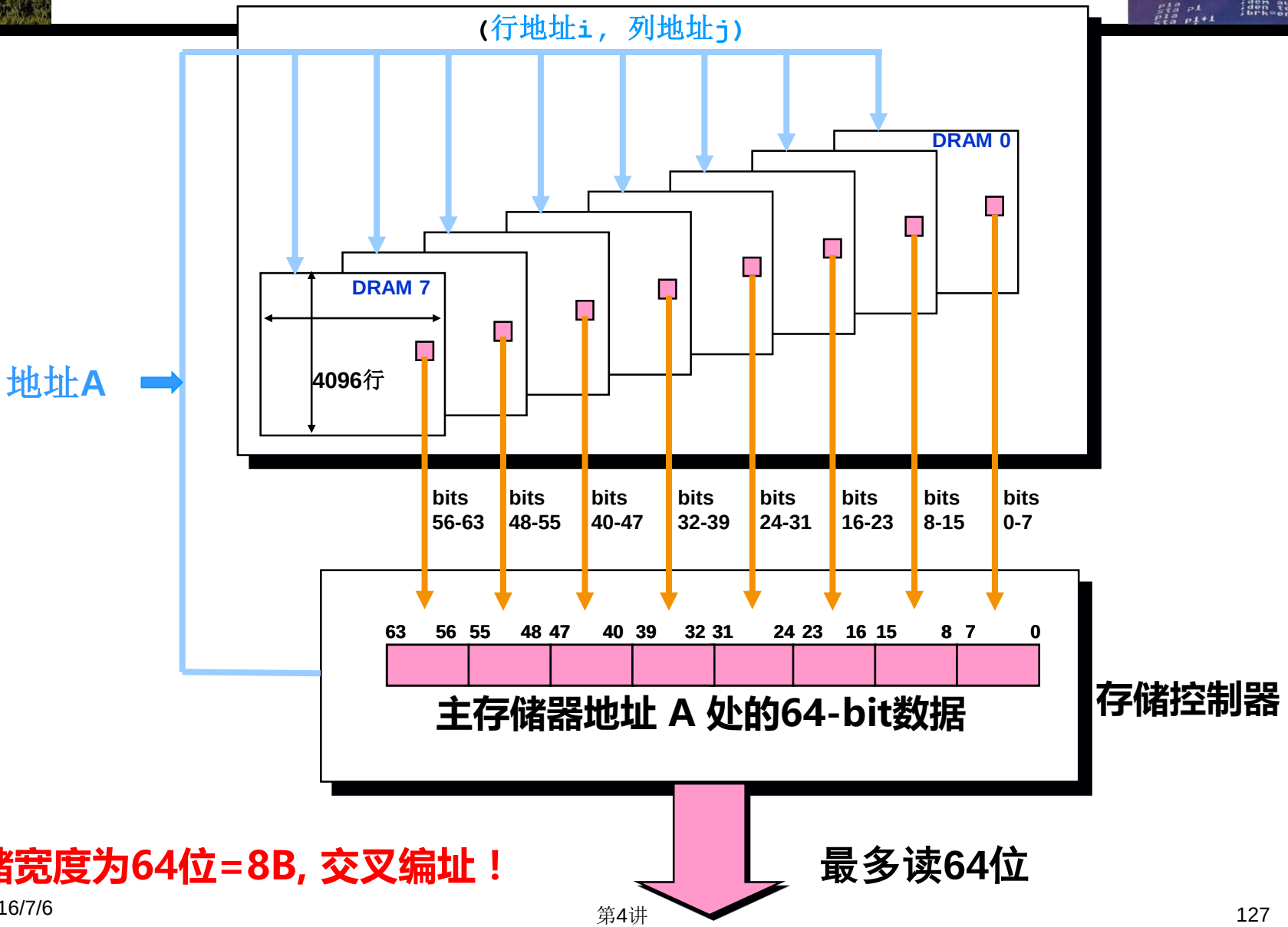
结构变量首地址按4字节边界对齐





只要SD首址按4B边界对齐，所有字段都能按要求对齐

主存储器的结构

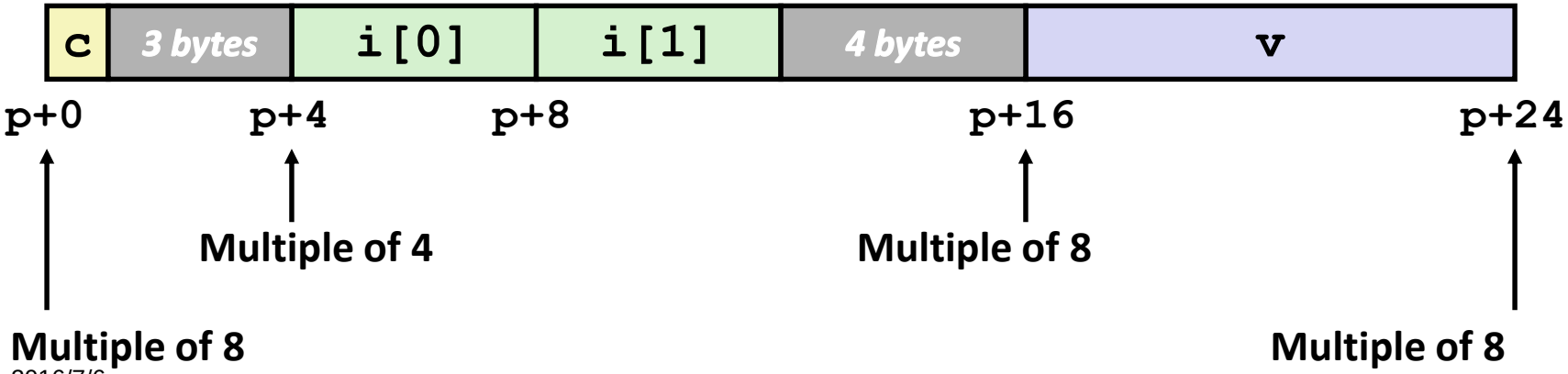


存储宽度为64位=8B，交叉编址！

在结构中满足对齐

- 结构中的偏移量
  - 必须满足元素的对齐要求
- 整个结构放置
  - 每个结构有对齐要求K
    - 成员中最大的对齐要求
  - 起始地址& 结构长度必须是K的倍数
- 例子 (在Windows或x86-64下):
  - K = 8, 由于double 元素

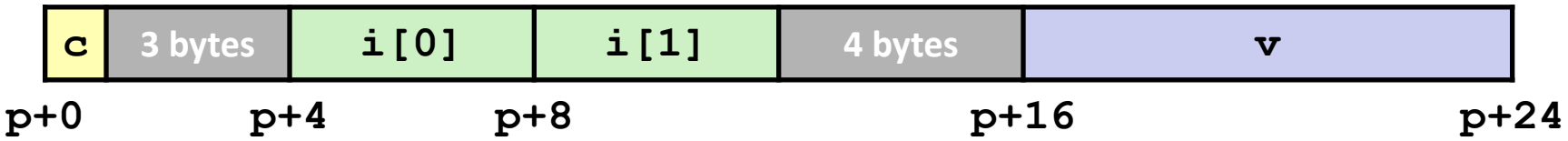
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



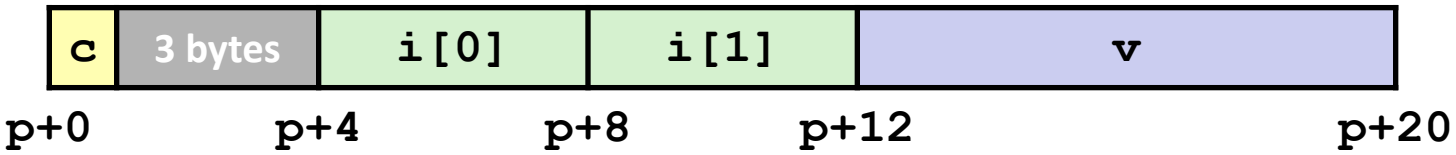
Linux vs. Windows

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

- x86-64 or IA32 Windows :
  - K = 8, 由于double 元素



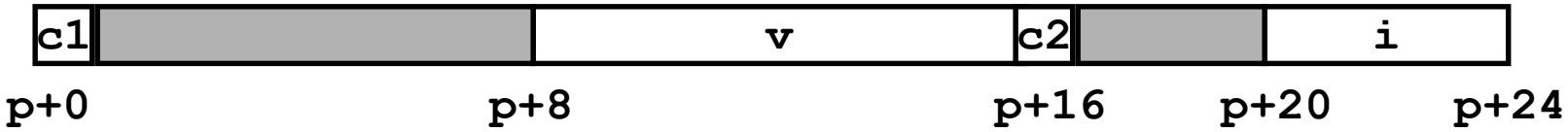
- IA32 Linux:
  - K = 4; double与4字节数据类型同样对待



结构内的元素顺序

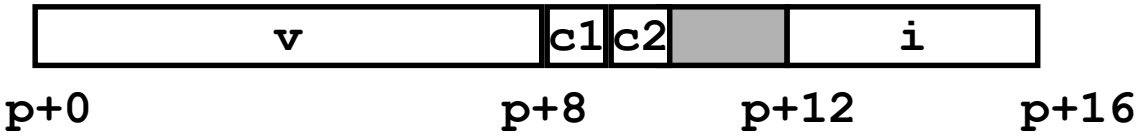
```
struct S4 {
   char c1;
   double v;
   char c2;
   int i;
} *p;
```

在Windows中10个字节浪费掉了




```
struct S5 {
   double v;
   char c1;
   char c2;
   int i;
} *p;
```

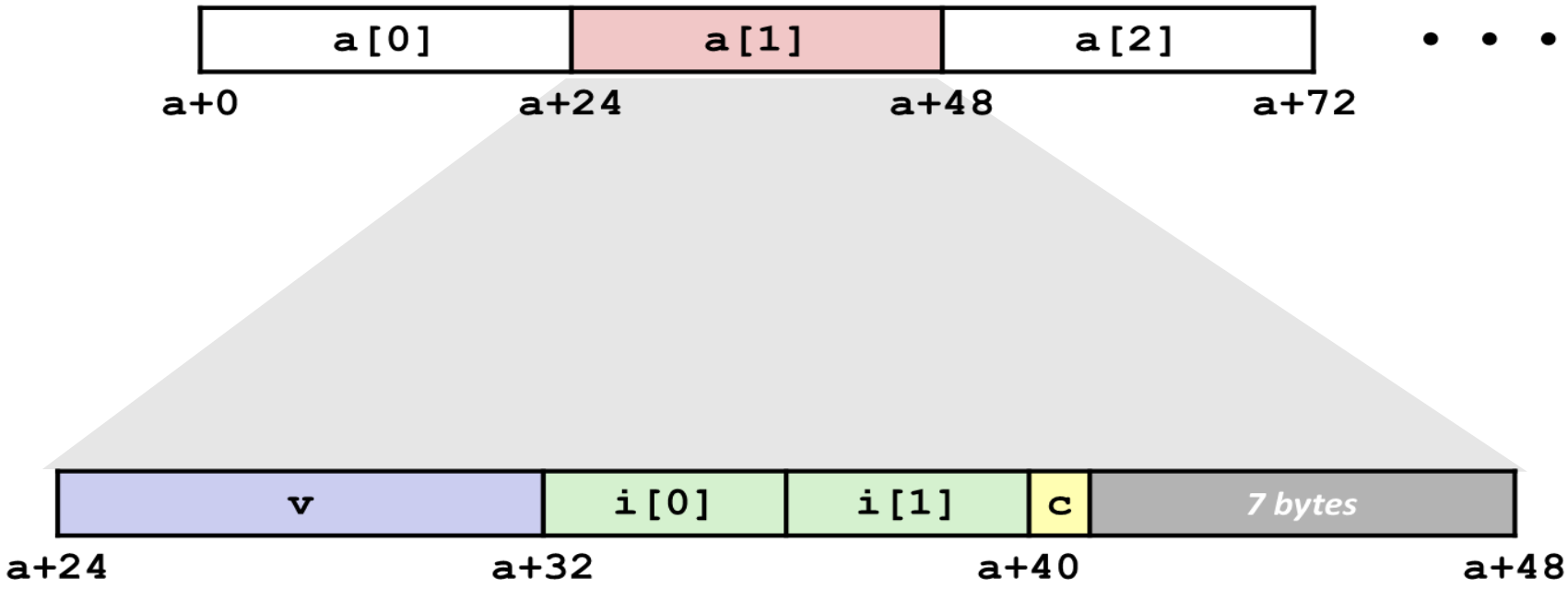
2个字节浪费掉了



结构数组

- 全部结构长度是 K 倍数
- 每一个元素满足对齐要求

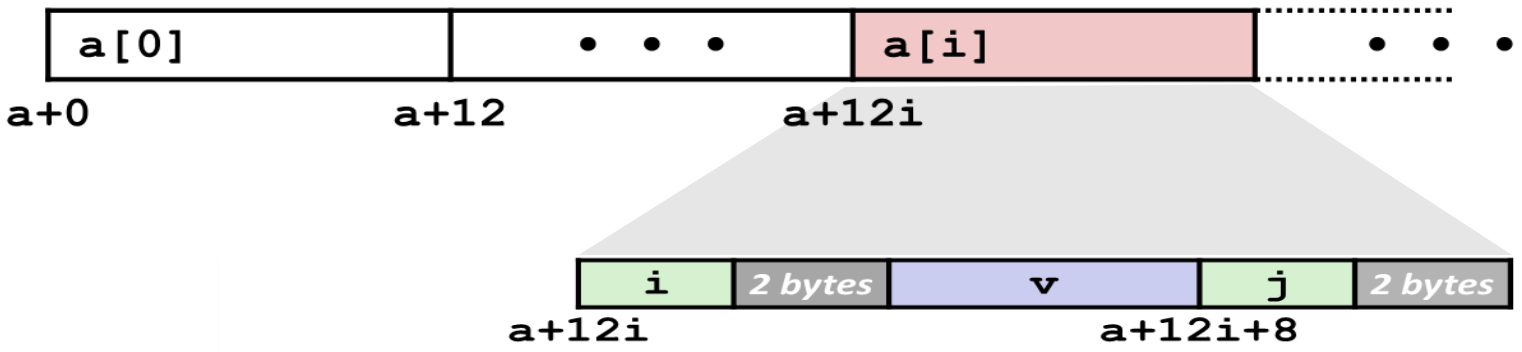
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



数组内访问元素

- 计算数组偏移量 12*I
  - **sizeof(S3)**,包括对齐空间
- 元素j在结构内的偏移量为8
  - 汇编器给出偏移量为：a+8
    - 在链接时解析为实际值

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```




```
short get_j(int idx)
{
  return a[idx].j;
}
```

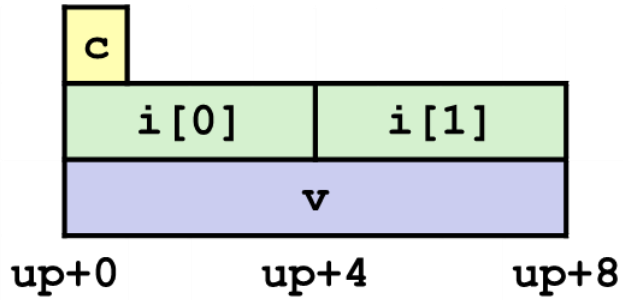
```
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```

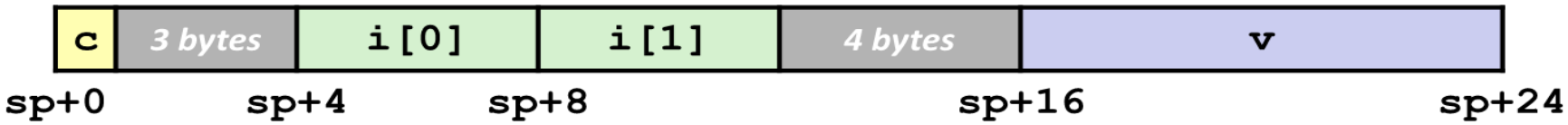
联合分配

- 按照最大的元素分配空间
- 元素重叠
- 一次只能使用一个域

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```

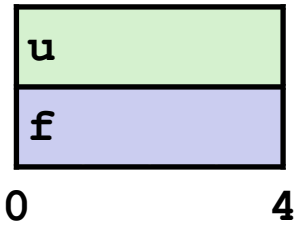
```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```





使用联合来访问位模式

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```




```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

- 直接访问float的位的形式
- `bit2float` 按照给定的位形式创建一个float
  - NOT the same as `(float) u`
- `float2bit` 由一个float生成相应的位的形式
  - NOT the same as `(unsigned) f`

联合体数据的分配和访问

联合体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

```
union uarea {
    char  c_data;
    short s_data;
    int   i_data;
    long  l_data;
};
```

IA-32中编译时，long和int长度一样，故uarea所占空间为4个字节。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11个字节，对齐的话则占用更多。

- 通常用于特殊场合，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。
- 但有时会得不偿失，可能只会减少少量空间却大大增加处理复杂性。

联合体数据的分配和访问

- 还可实现对相同位序列进行不同数据类型的解释

```
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，因而传递过来的实参是float型数据，赋值给非静态局部变量（联合体变量成员）

过程体为:

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

可优化掉！

将存放在地址R[ebp]+8处的入口参数 f 送到EAX（返回值）

问题：float2unsign(10.0)=? $2^{30}+2^{24}+2^{21}=1092616192$

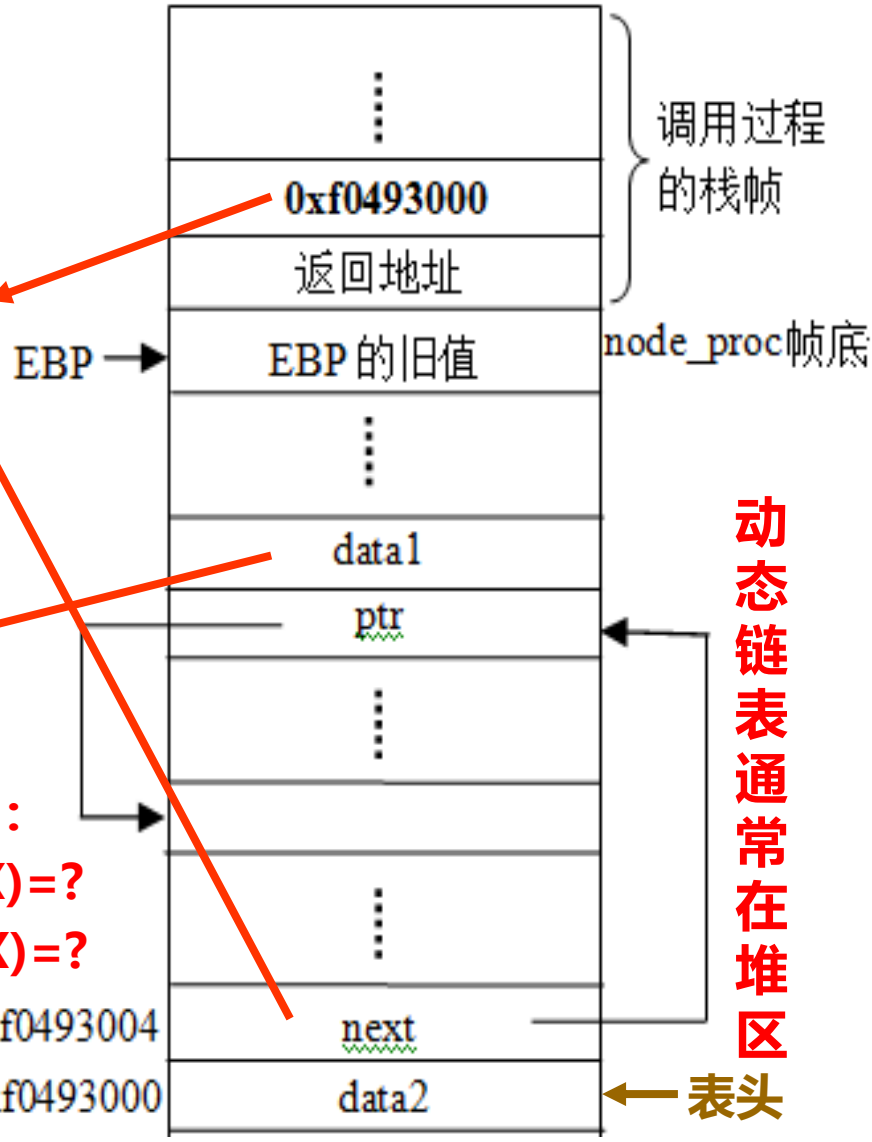
从该例可看出：机器级代码并不区分所处理对象的数据类型，不管高级语言中将其说明成float型还是int型或unsigned型，都把它当成一个0/1序列来处理。

联合体数据的分配

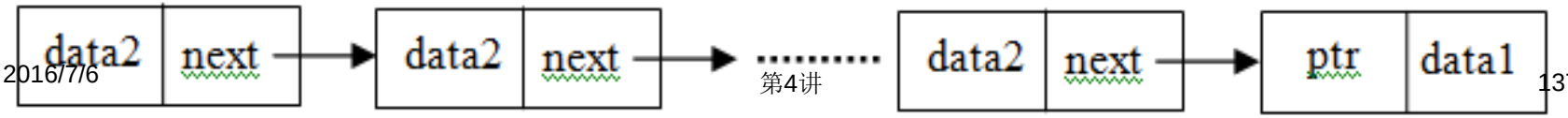
- 利用嵌套可定义链表结构

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

```
movl  8(%ebp), %ecx
movl  4(%ecx), %edx
movl  (%edx), %eax
movl  (%eax), %eax
addl  (%ecx), %eax
movl  %eax, 4(%edx)
```




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



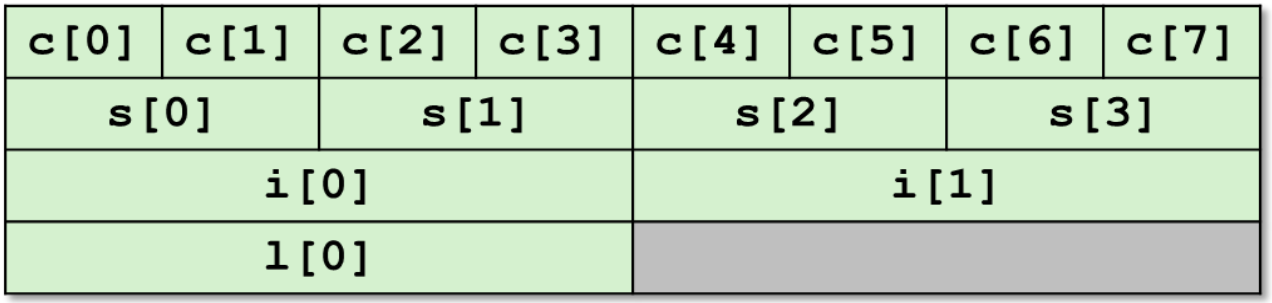
重新考虑字节序

- 思想
  - Short/long/quad words在存储器中分别占用连续2/4/8字节
  - 哪位表示最高 (最低)有效位?
  - 在不同机器间交换二进制数据可能会引发问题
- 大端法
  - 最高有效位字节在低地址
  - PowerPC, Sparc
- 小端法
  - 最低有效位字节在低地址
  - Intel x86, Alpha

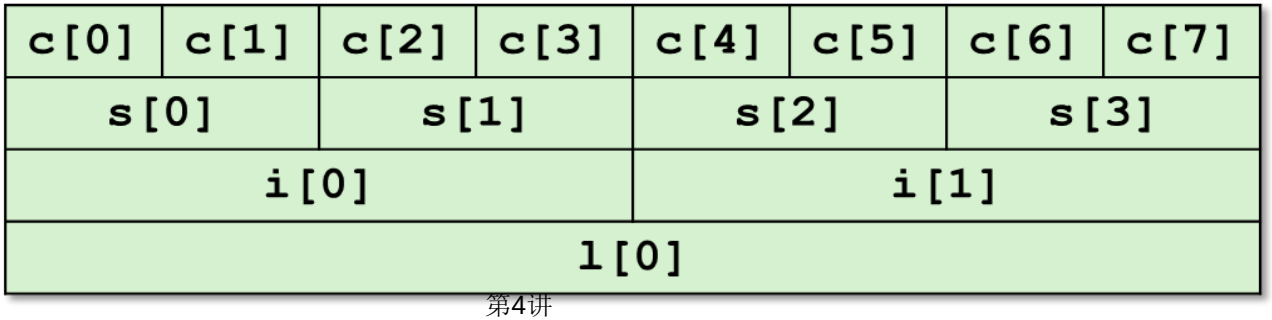
字节序例子

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

32-bit



64-bit

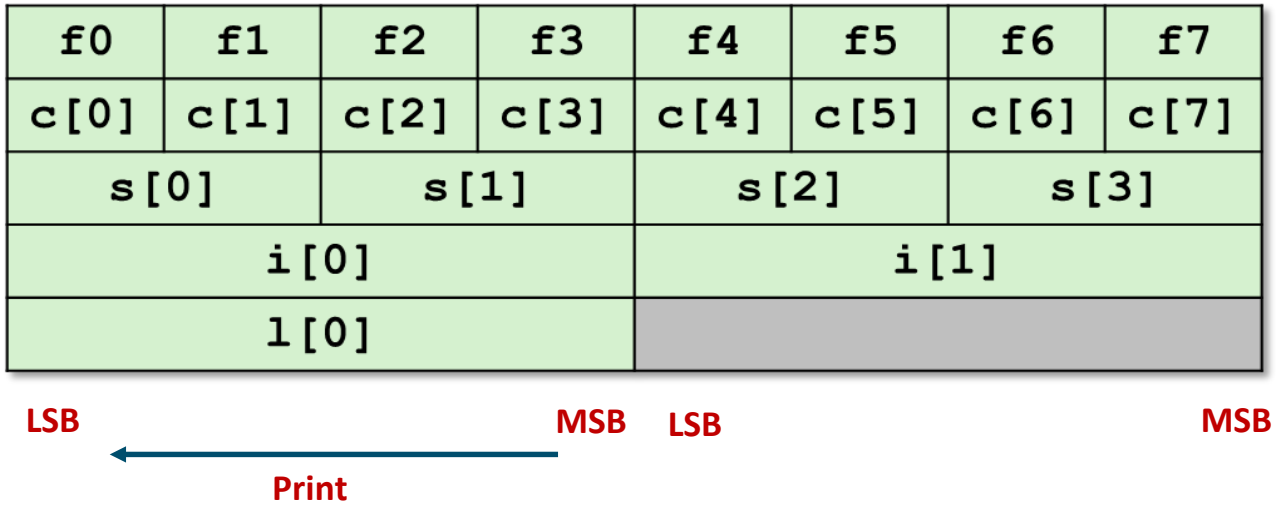


字节序例子(续)

```
int j;
for (j = 0; j < 8; j++)
    dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 == [0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

IA32 字节序

小端法

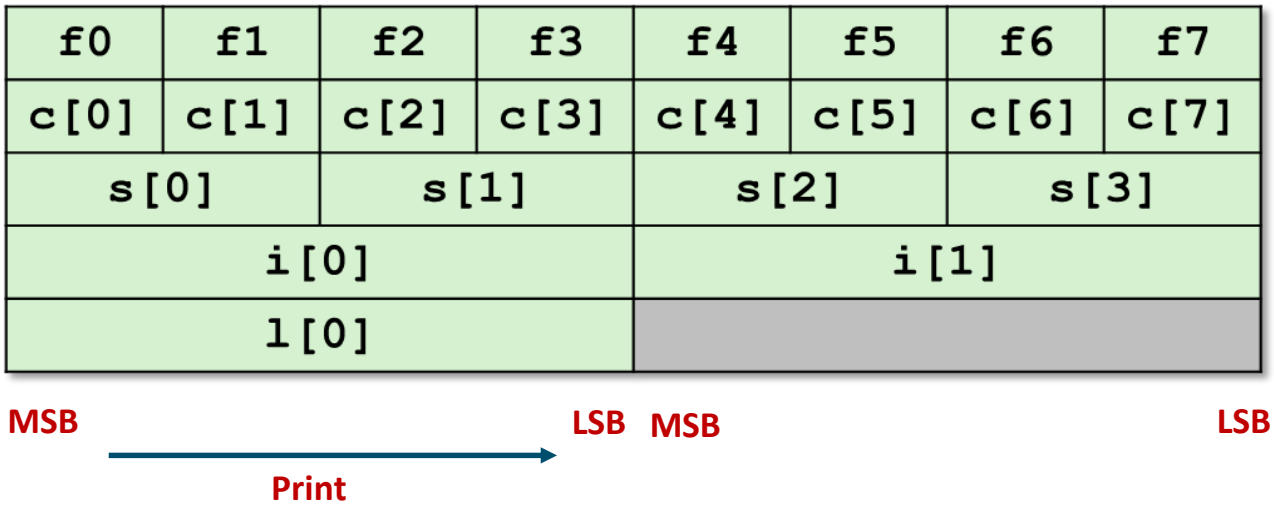


Output:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf3f2f1f0]
```

Sun机器字节序

大端法

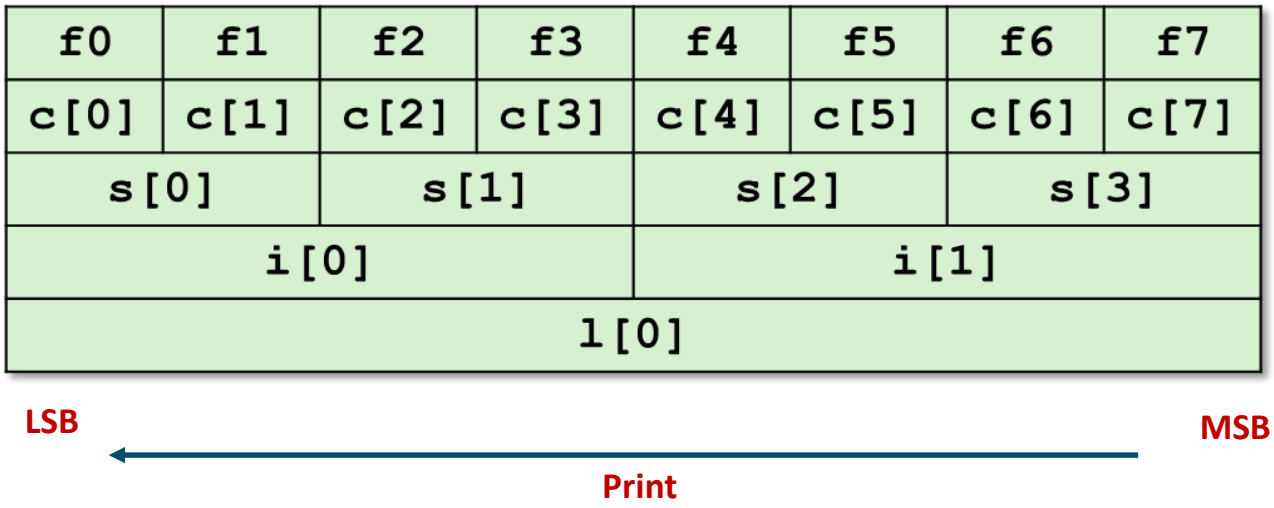


Output on Sun:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```

X86-64 字节序

小端法



Output on x86-64:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

总结

- C中数组
  - 连续的存储空间
  - 指向最开始元素
  - 无边界检测
- 结构
  - 按照声明顺序分配字节
  - 为了满足对齐在中间和最后填充空隙
- 联合
  - 叠加声明
  - 绕过类型系统的一种方法

作业

- 练习：3.32，3.33，3.34, 3.36，3.37，3.39，3.40，3.41
- 作业：3.60，3.62，3.63，3.64，3.66，3.67