过程调用的机器级表示

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

过程调用的机器级表示

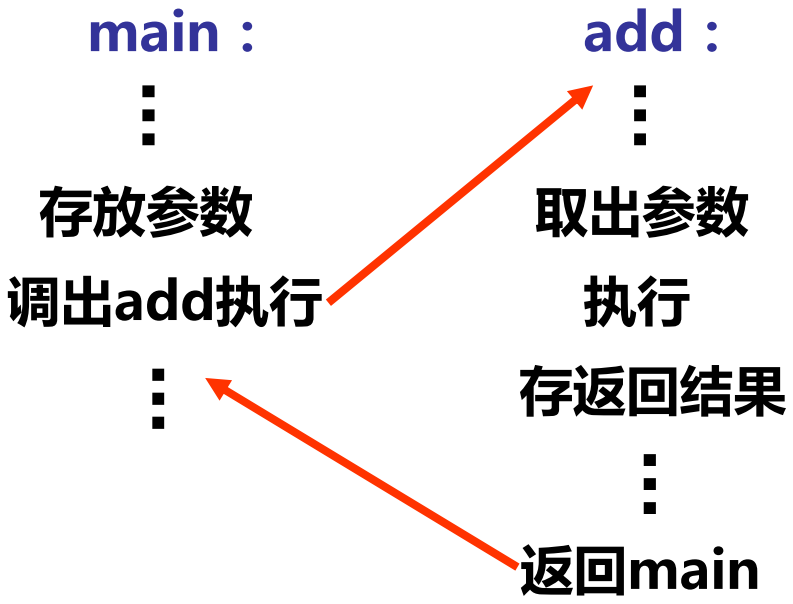
- 以下过程（函数）调用对应的机器级代码是什么？
- 如何将t1(125)、t2(80)分别传递给add中的形式参数x、y
- add函数执行的结果如何返回给caller?

```
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

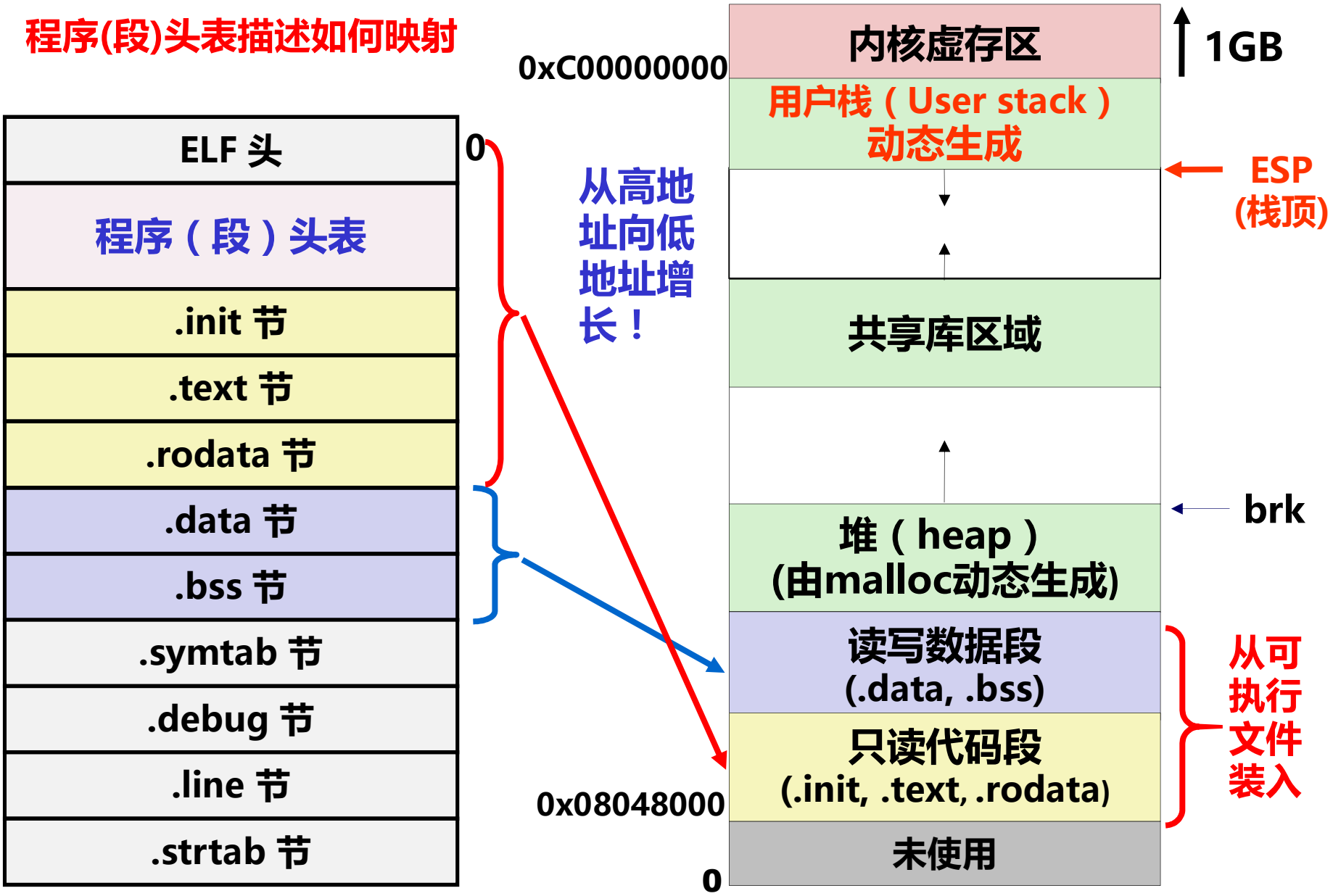
main



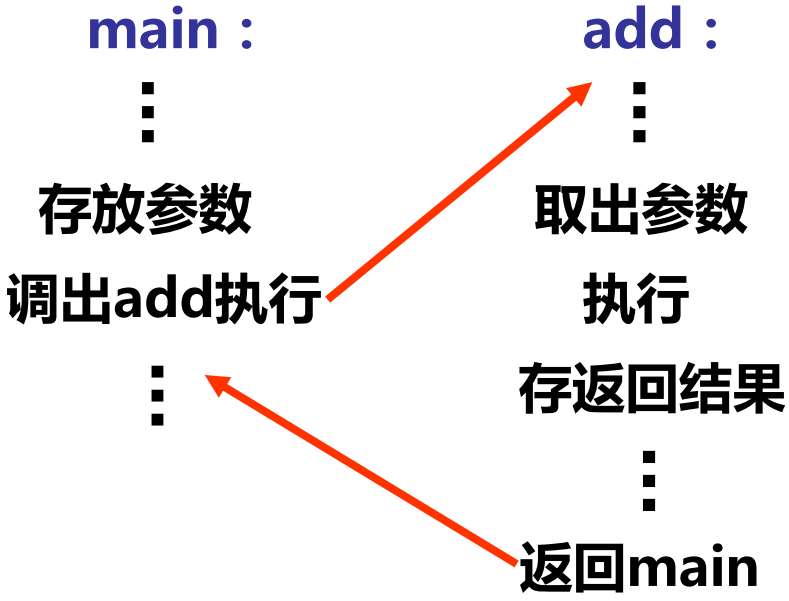
参数通过栈（stack）来传递！

栈（stack）在哪里？

可执行文件的存储器映像



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

想象：妈妈和你做菜时共用一套盘子的情况。

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**

想象一下，共用同一套盘子做菜的情况！

  - **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。

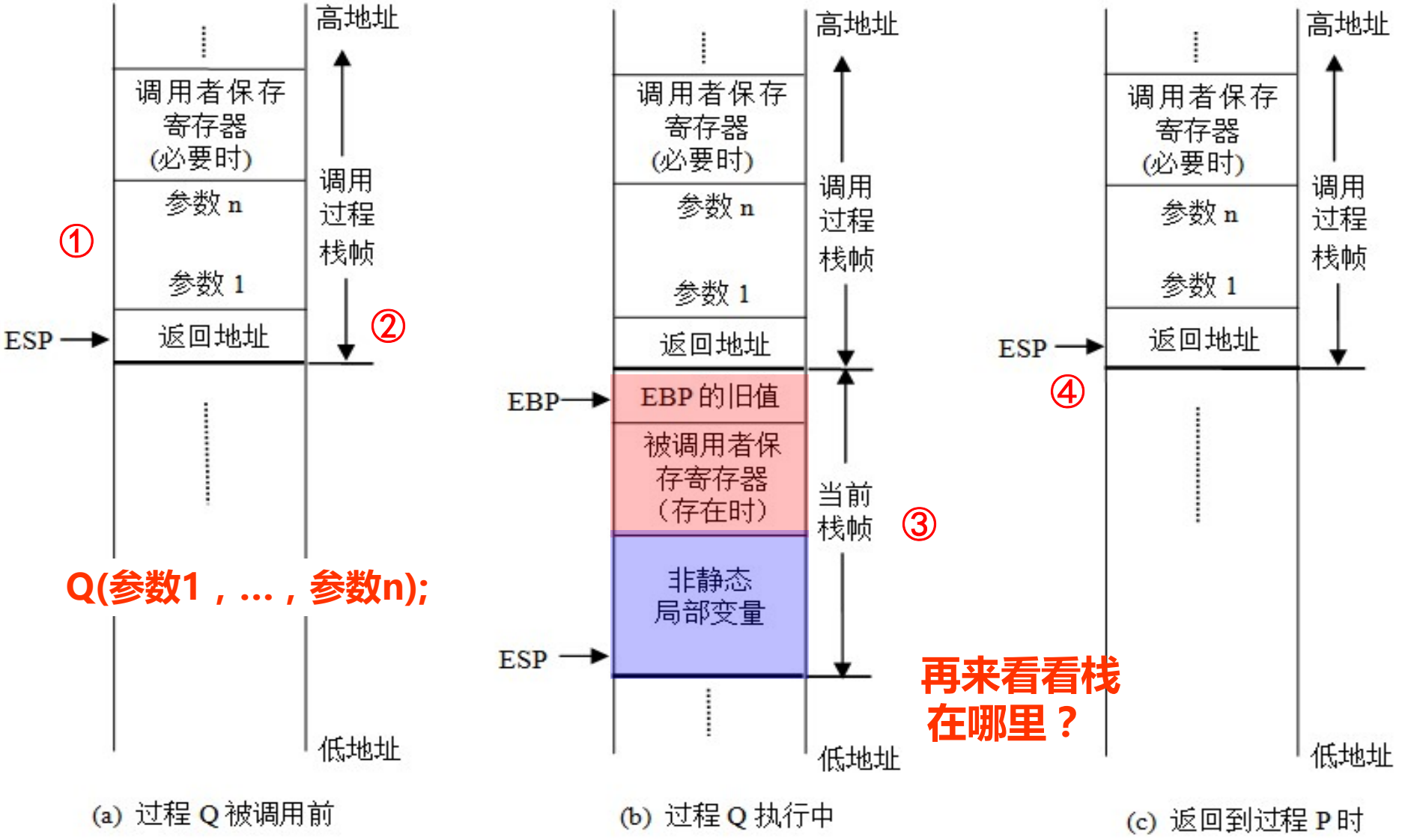
  - **EBP和ESP分别是帧指针寄存器和栈指针寄存器，分别用来指向当前栈帧的底部和顶部。**

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？

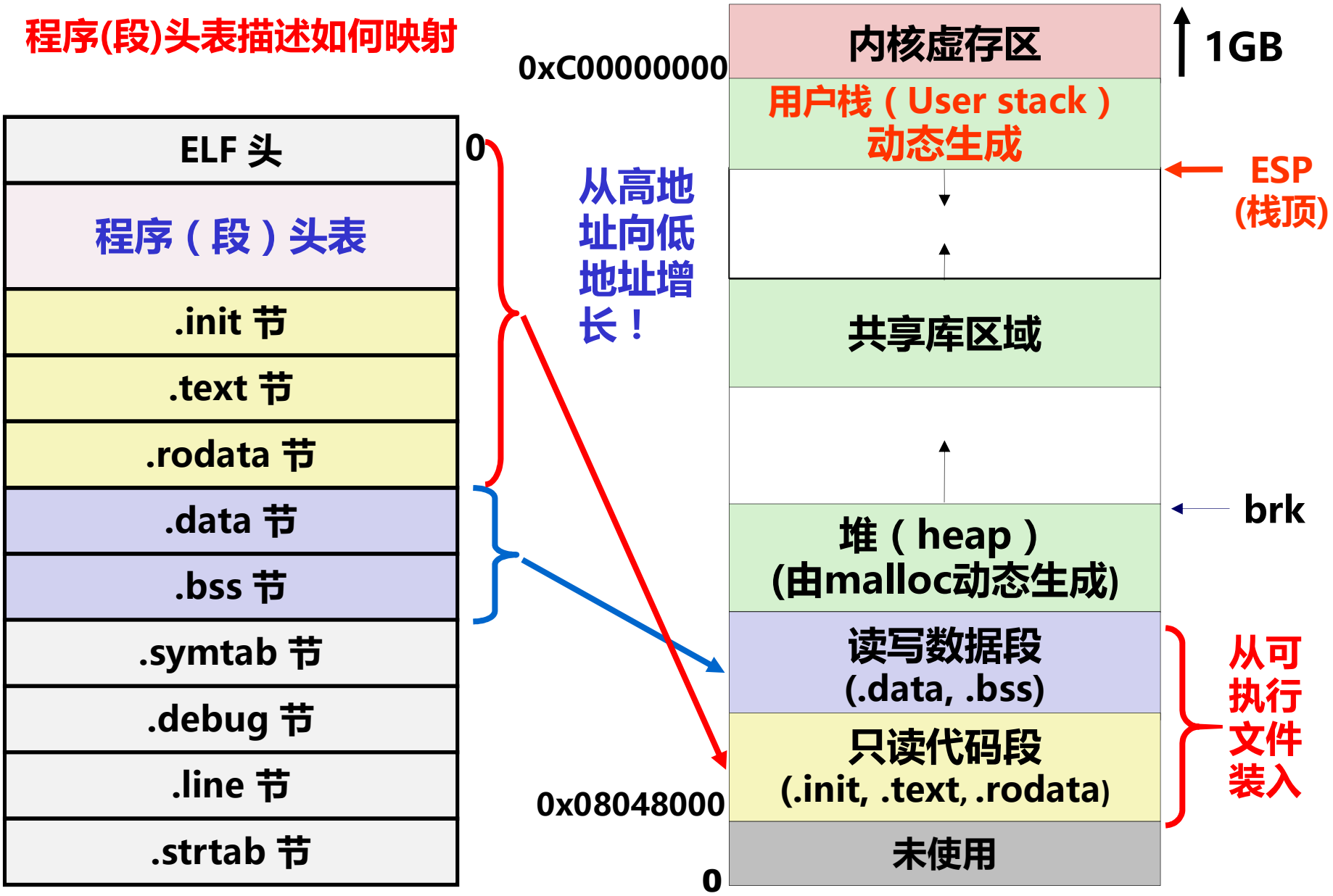
EAX、ECX、EDX！

过程调用的机器级表示

- **过程调用过程中栈和栈帧的变化 (Q为被调用过程)**



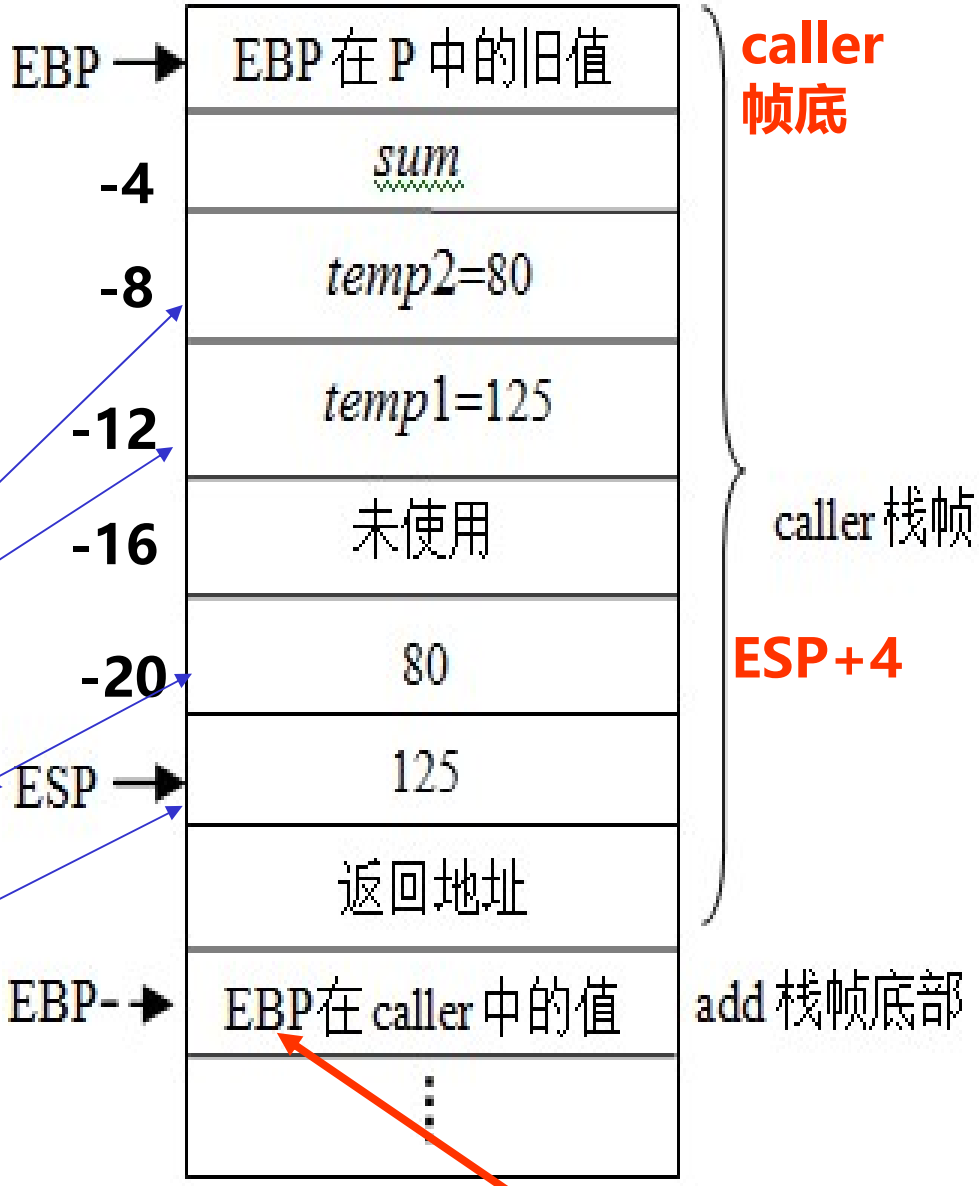
可执行文件的存储器映像



```
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



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

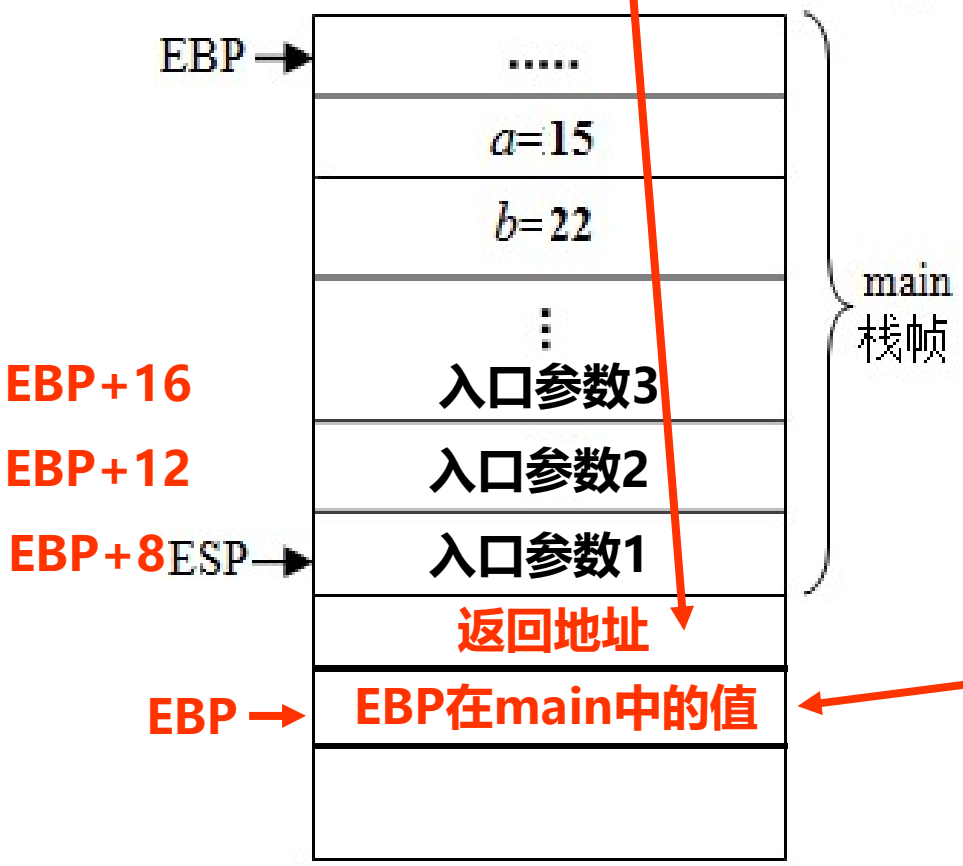
- 一个C过程的大致结构如下：
  - 准备阶段
    - 形成帧底：push指令 和 mov指令
    - 生成栈帧（如果需要的话）：sub指令 或 and指令
    - 保存现场（如果有被调用者保存寄存器）：mov指令
  - 过程（函数）体
    - 分配局部变量空间，并赋值
    - 具体处理逻辑，如果遇到函数调用时
      - 准备参数：将实参送栈帧入口参数处
      - CALL指令：保存返回地址并转被调用函数
    - 在EAX中准备返回参数
  - 结束阶段
    - 退栈：leave指令 或 pop指令
    - 取返回地址返回：ret指令

入口参数的位置

```
movl  参数3，8(%esp)
.....
movl  参数1, (%esp)
call   add
```

准备入口参数

R[esp]←R[esp]-4
M[R[esp]]←返回地址
R[eip]←add函数首地址



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：

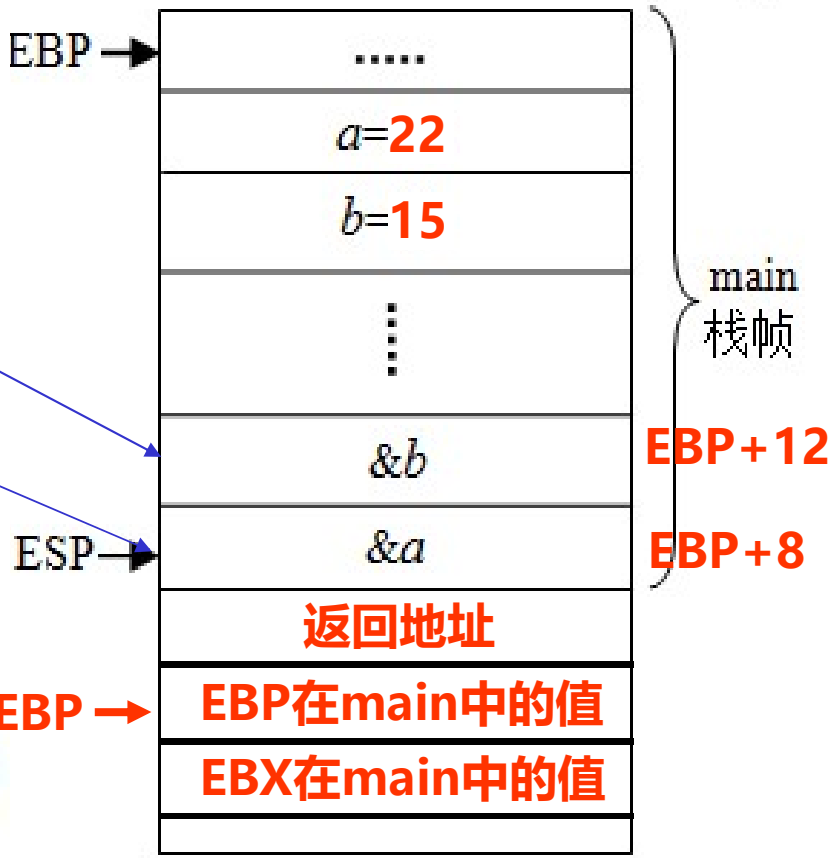
a=15　b=22
a=15　b=22

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 → EBP在main中的值

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

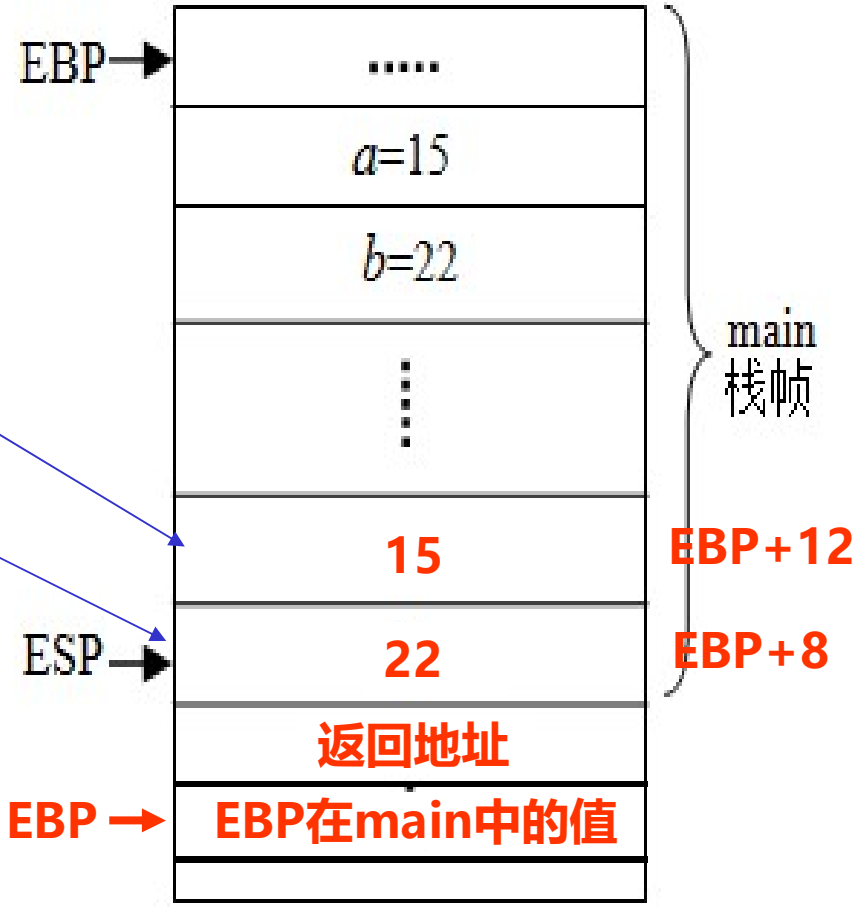
过程调用参数传递举例

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

M[R[ebp]+8] ← R[eax] =22

M[R[ebp]+12] ← R[edx] =15

局部变量a和b没有交换，
交换的仅是入口参数

过程调用

```
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  }
```

test

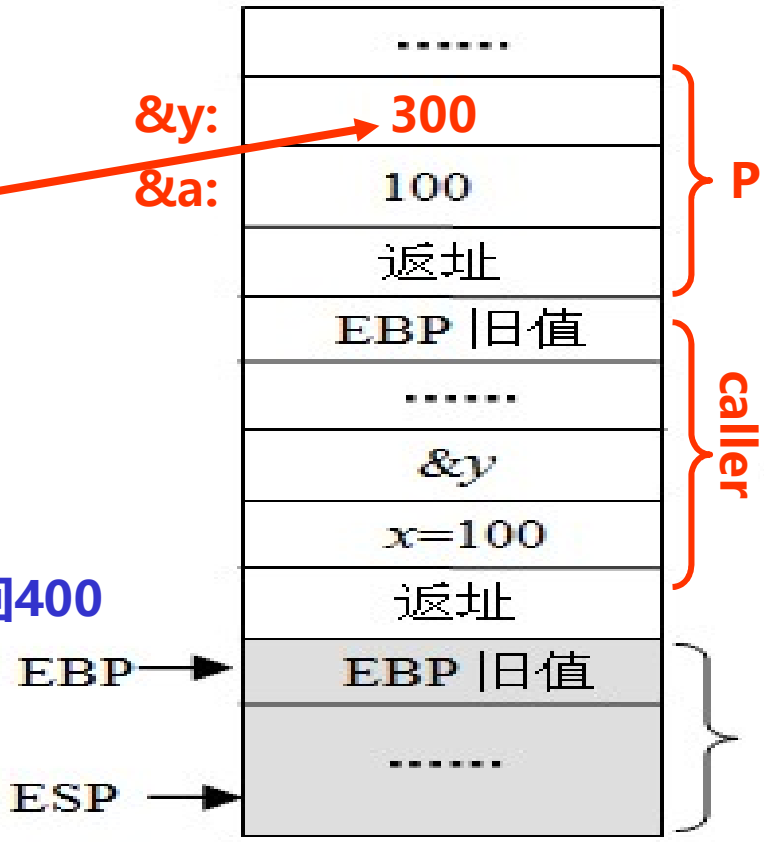
caller

P

100 200

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

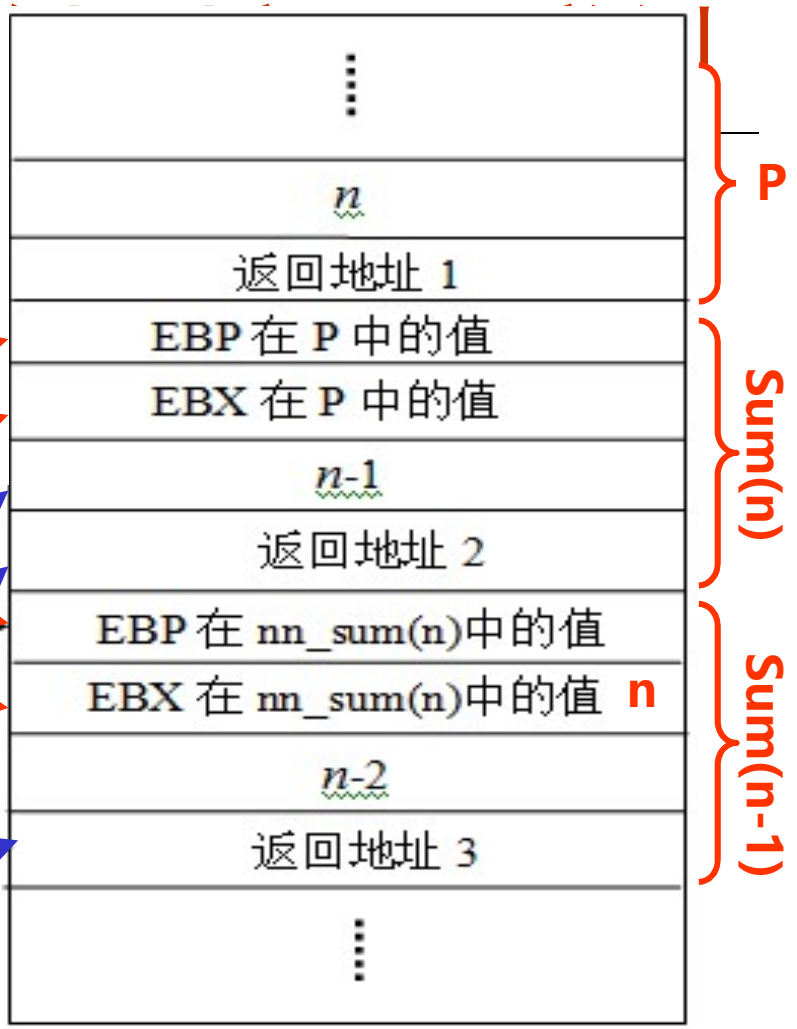
R[ebx]←n

R[eax]←0

if (n≤0) 转L2

R[eax]←n-1

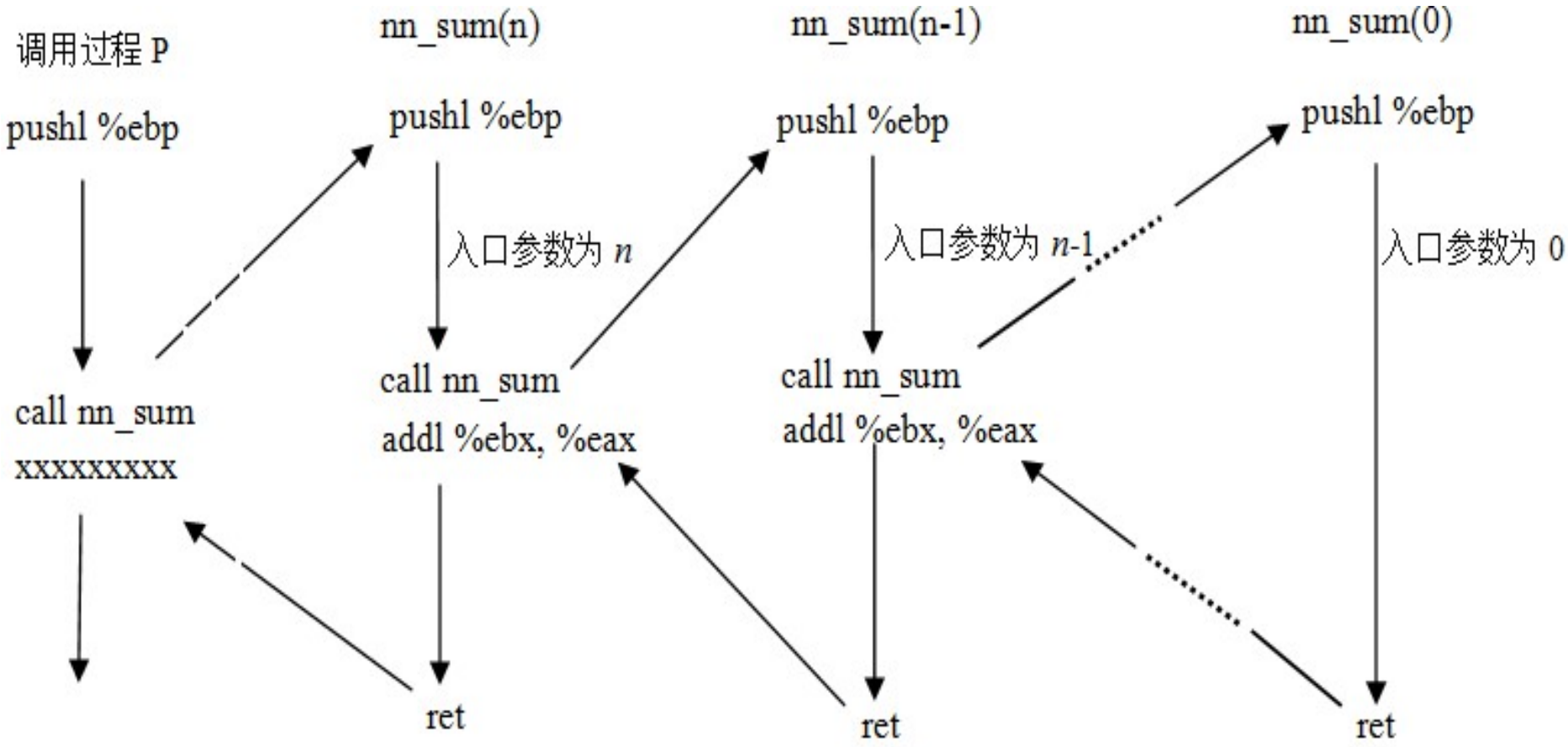
R[eax] ← 0+1+2+…+(n-1)+n



每次递归调用都会增加一个栈帧，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



过程功能由过程体实现，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) → 2.0000061035156
fun(4) → 3.14, 然后存储保护错

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？

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

```
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
```

a[i]=1073741824;

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

return d[0];

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