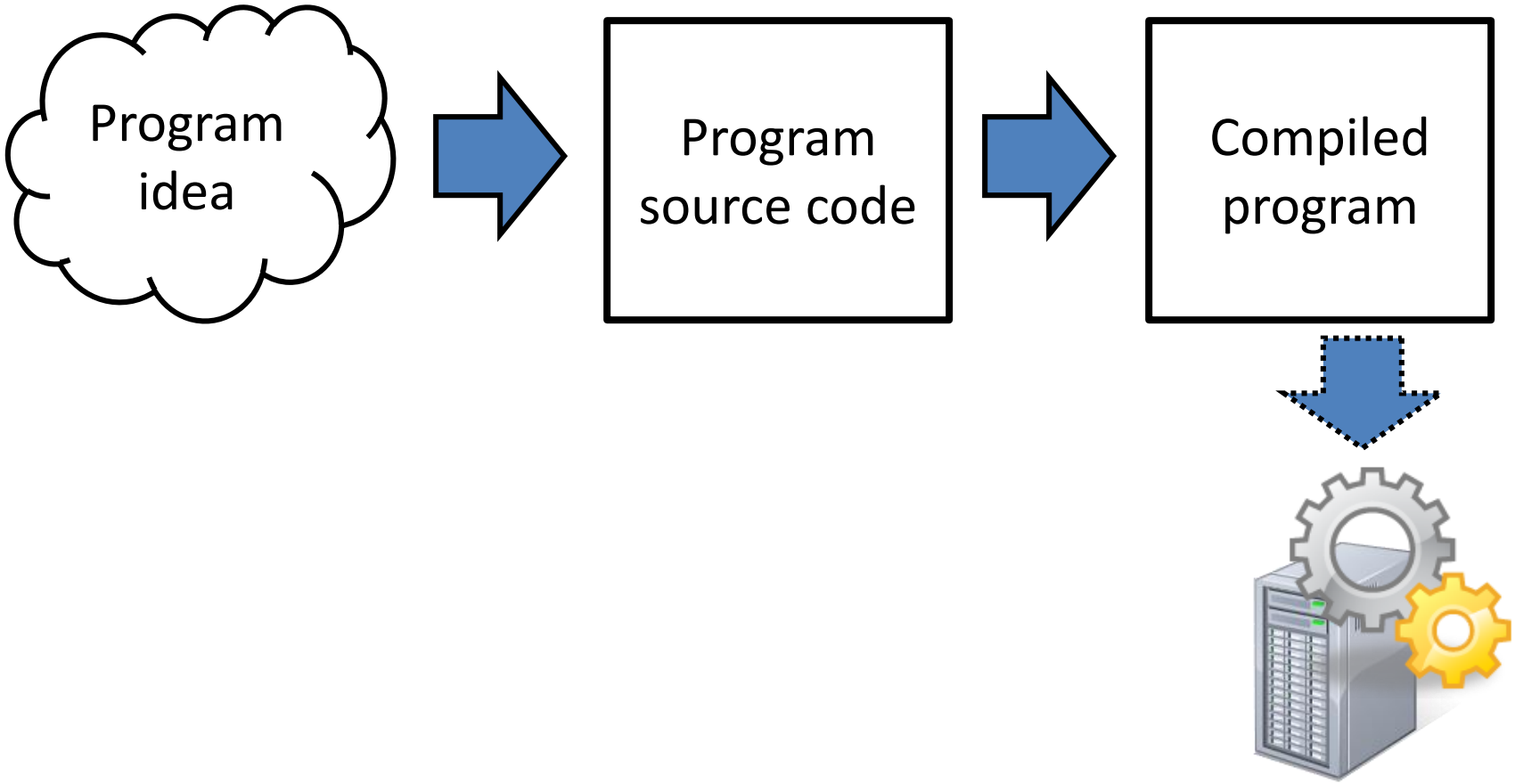
第2讲

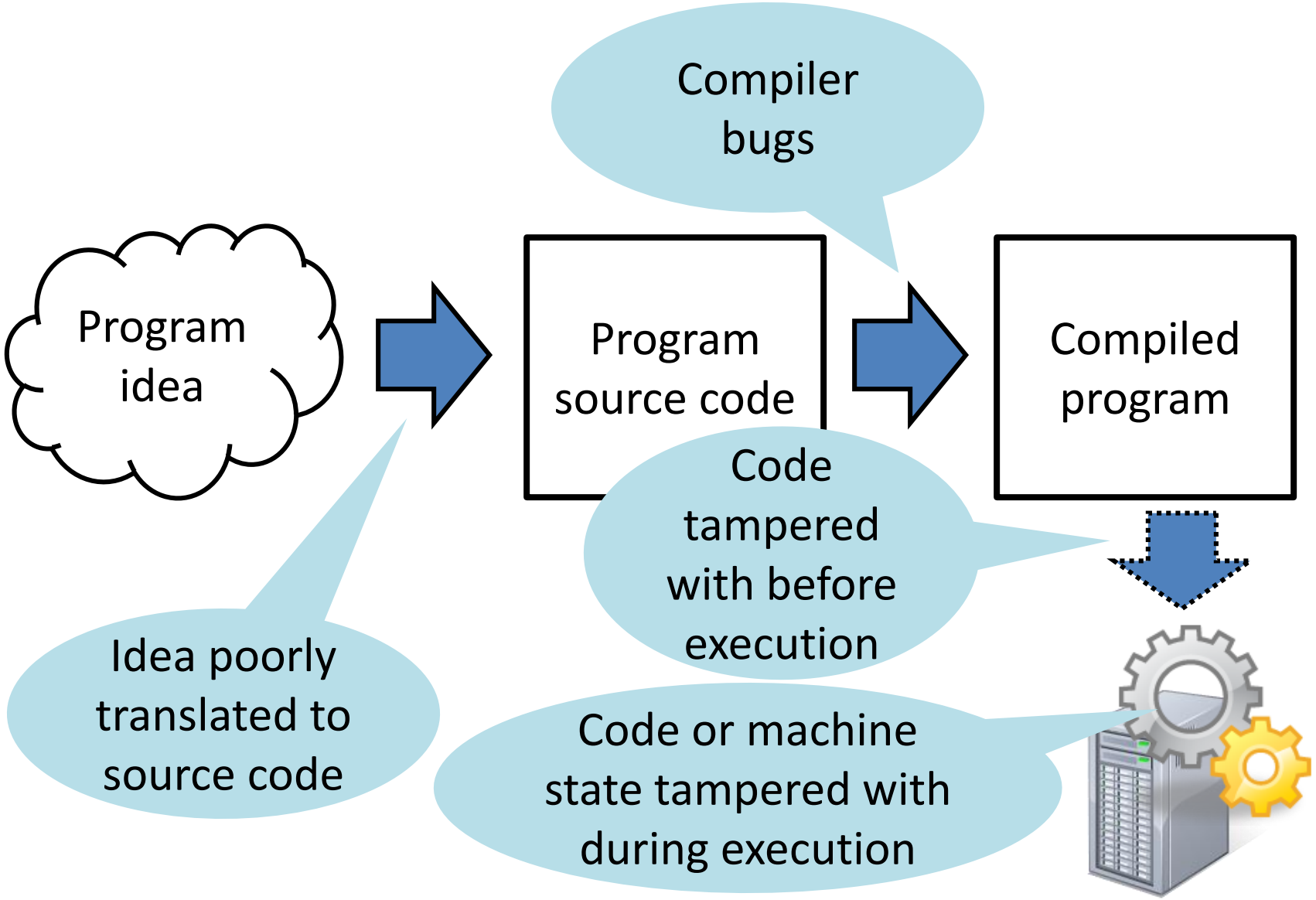
软件安全（1）

系统模型：从源代码到执行

从想法到执行



为什么程序有(安全)bug



Compiler Bugs

```
1  static int g[1];
2  static int *p = &g[0];
3  static int *q = &g[0];
4
5  int foo (void) {
6    g[0] = 1;
7    *p = 0;
8    *p = *q;
9    return g[0];
10 }
```

foo的返回值？

http://gcc.gnu.org/bugzilla/show_bug.cgi?id=42952

Finding and Understanding Bugs in C Compilers
https://www.cs.utah.edu/~regehr/papers/pldi11-preprint.pdf

Compiler Bugs

```
1 int x = 4;
2 int y;
3
4 void foo (void) {
5   for (y = 1; y < 8; y += 7) {
6     int *p = &y;
7     *p = x;
8   }
9 }
```

foo返回时y的值？

http://gcc.gnu.org/bugzilla/show_bug.cgi?id=43360

程序的编译和执行

```
#include <stdio.h>
void answer(char *name, int x){
  printf(“%s, the answer is: %d\n”,
          name, x);
}
void main(int argc, char *argv[]){
  int x;
  x = 40 + 2;
  answer(argv[1], x);
}
```

42.c

```
void answer(char *name, int x){
  printf("%s, the answer is: %d\n",
         name, x);
}
void main(int argc, char *argv[]){
  int x;
  x = 40 + 2;
  answer(argv[1], x);
}
```

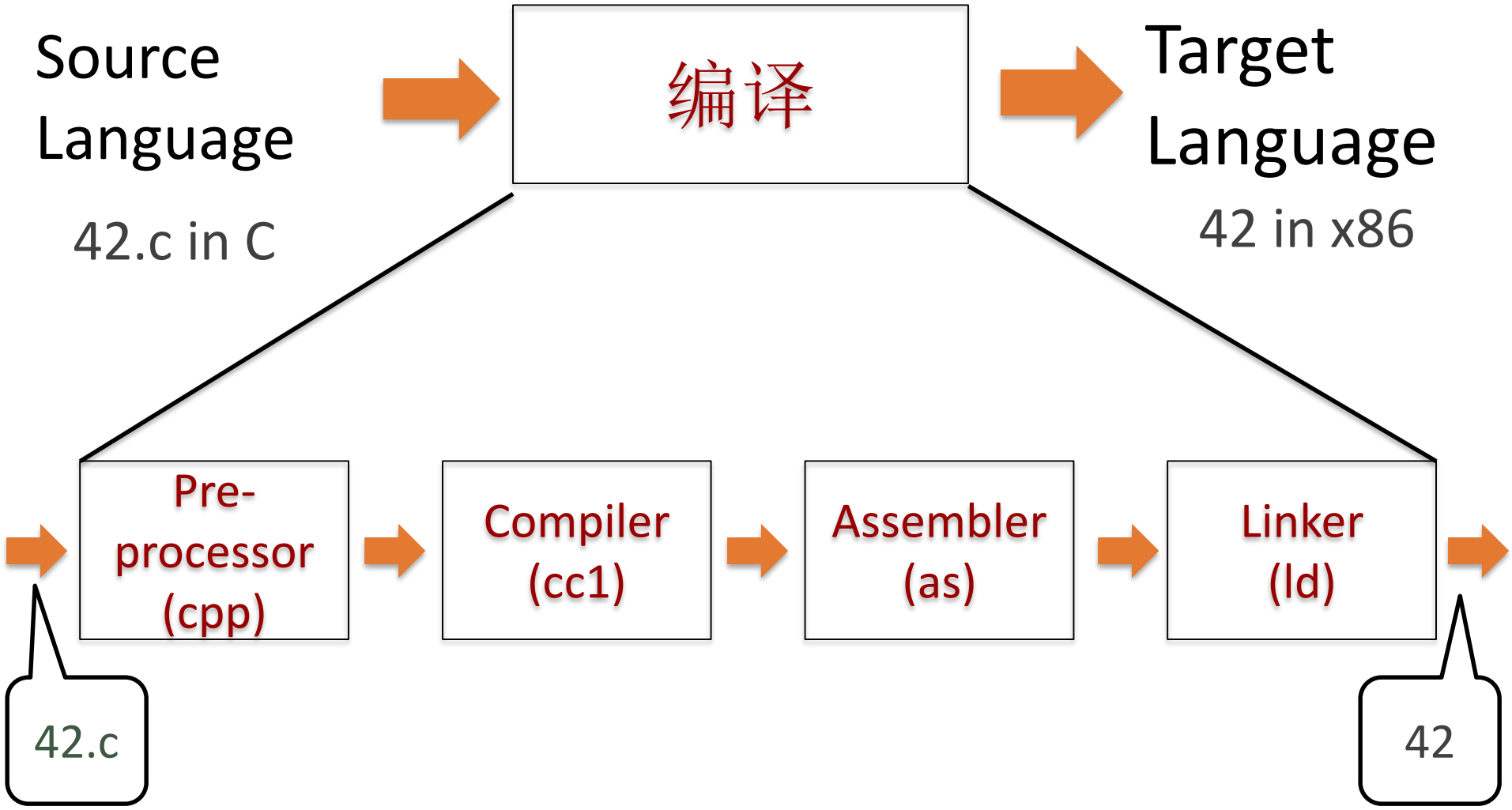
Compilation

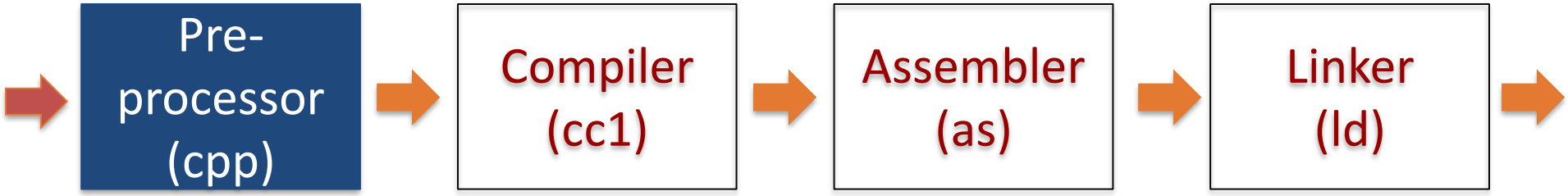
00110101
10101010
00101

David

David, the answer is 42

- 程序的编译
  - 以C语言程序为例
- 程序的执行


Source Language
42.c in C
编译
Target Language
42 in x86
Pre-processor (cpp)
Compiler (cc1)
Assembler (as)
Linker (ld)
42.c
42

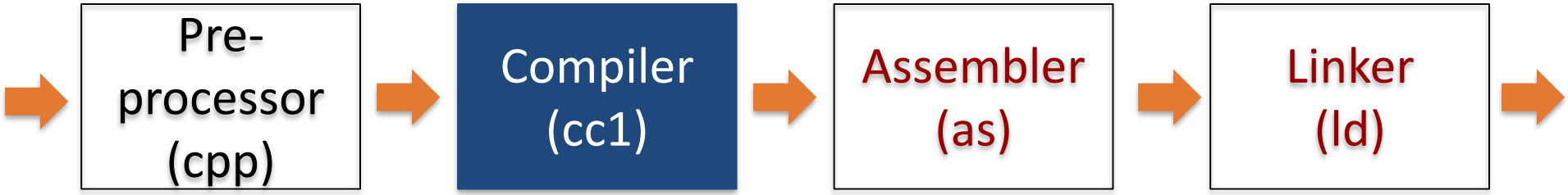


$ cpp

```
#include <stdio.h>
void answer(char *name, int x){
  printf("%s, the answer is: %d\n",
          name, x);
}
...
```

#include expansion

#define substitution



$ gcc -S

```
#include <stdio.h>
void answer(char *name, int x){
  printf("%s, the answer is: %d\n",
              name, x);
}
...
```

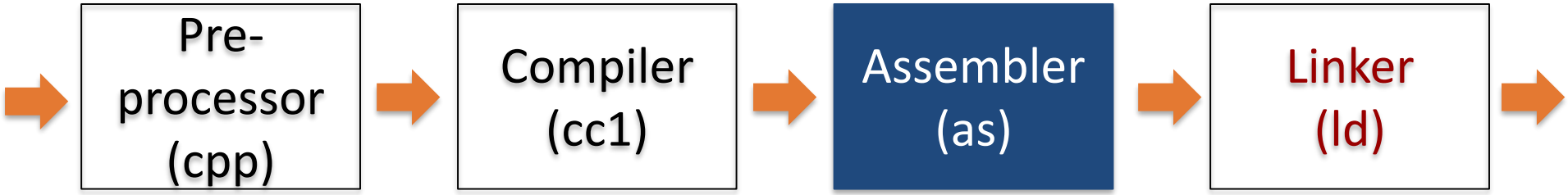
Creates Assembly

gcc -S 42.c outputs 42.s

```
…
answer:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        subl    $4, %esp
        pushl   12(%ebp)
        pushl   8(%ebp)
        push    $.LC0
        call    printf
        addl    $16, %esp
        leave
        ret
```

```
leave

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

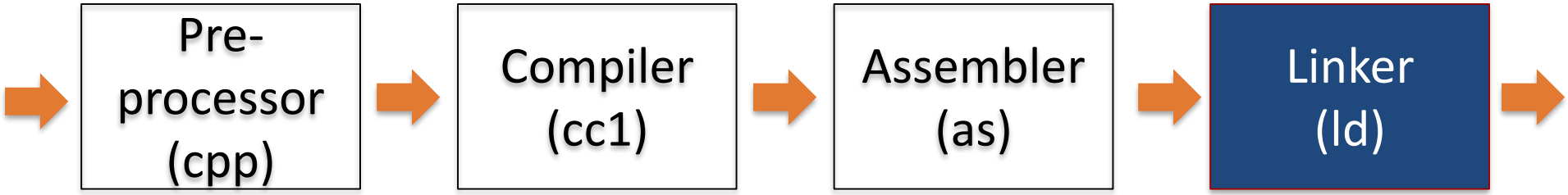


$ as <options>

```
...
answer:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        subl    $4, %esp
        pushl   12(%ebp)
        pushl   8(%ebp)
        push    $.LC0
        call    printf
        addl    $16, %esp
        leave
        ret
```

42.s

Creates object code

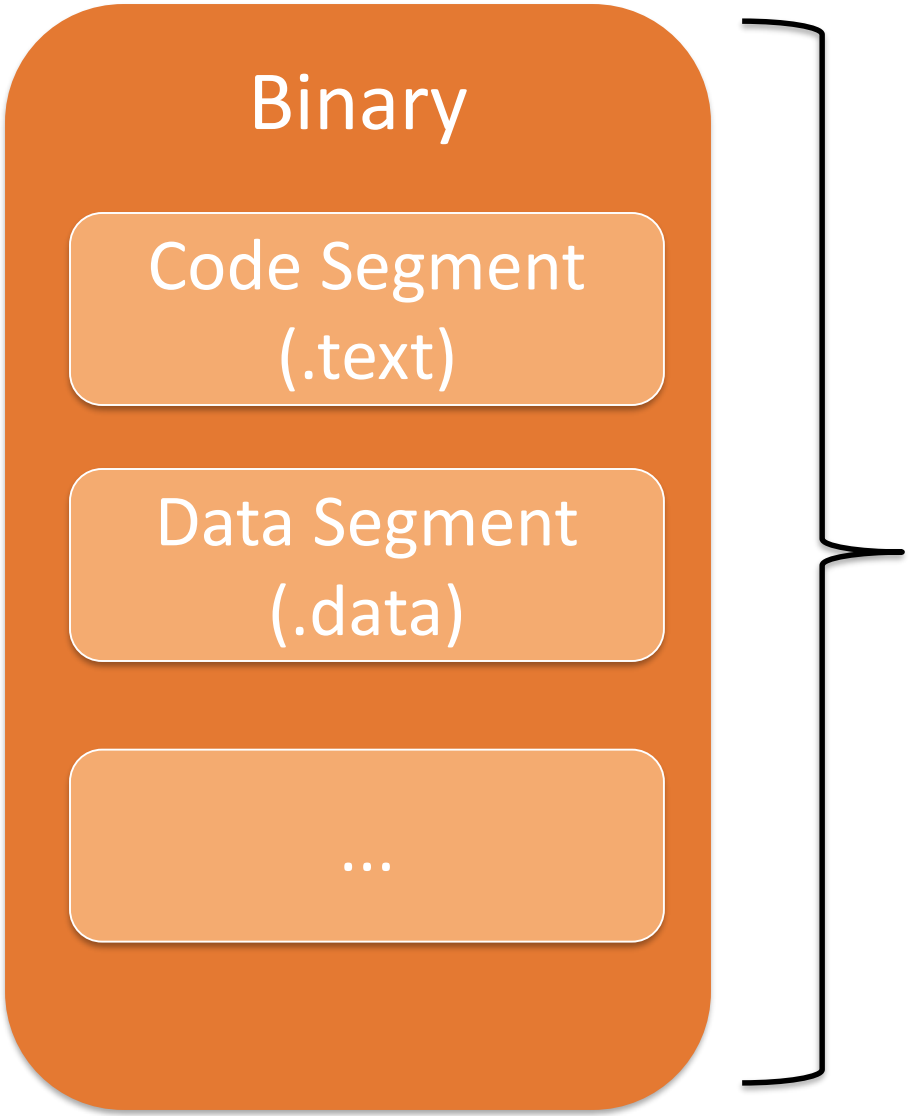


$ ld <options>

```
01011001010101011010101
10101010101010111111100
00110101011010100101011
01011110101001011000010
10111101
```

42.o

Links with other files and libraries to produce an executable

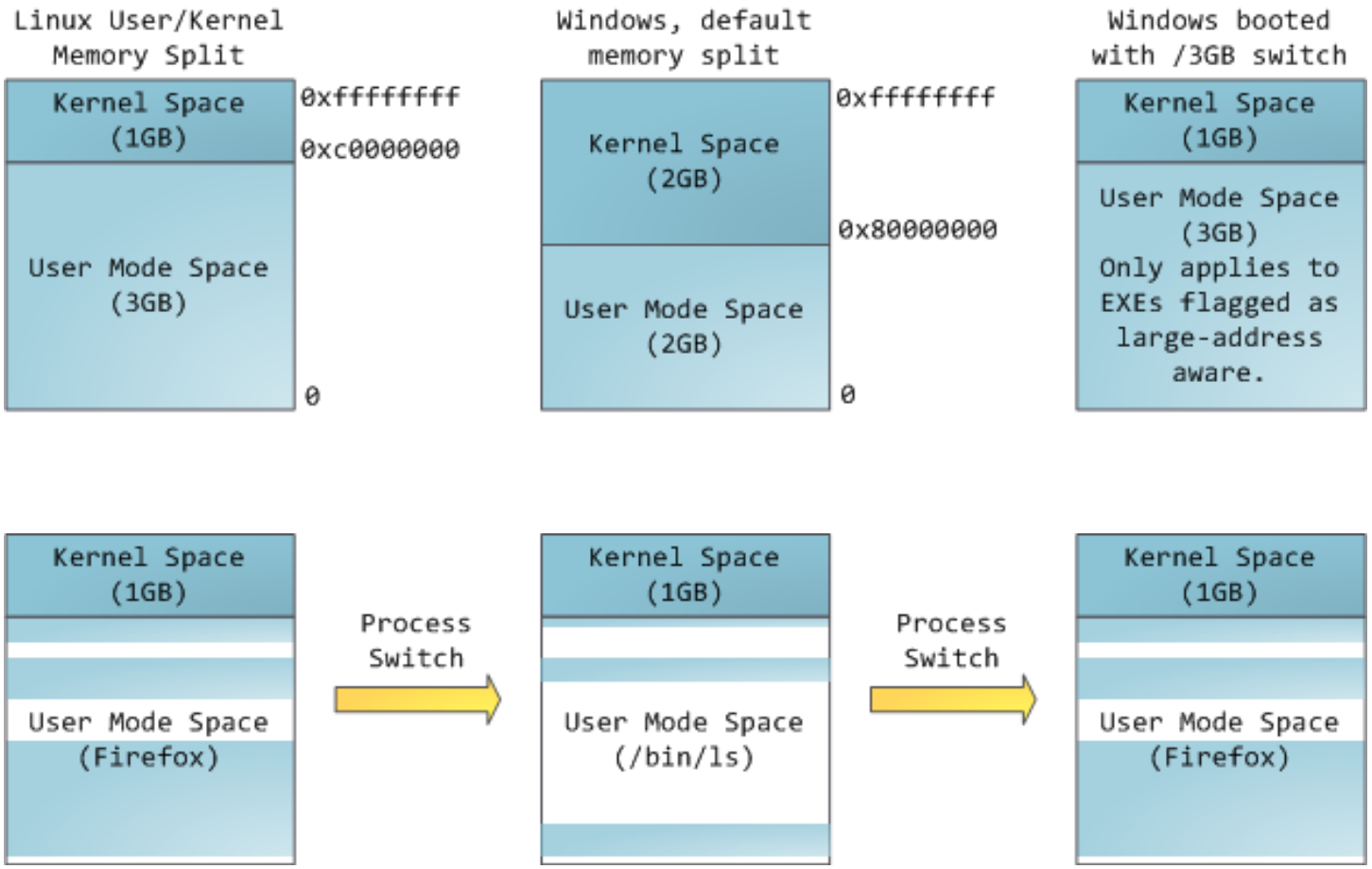


可执行文件**(Executable)**包括：

- 代码段
- 数据段

- 程序的编译
  - 以C语言程序为例
- 程序的执行

内存中的程序

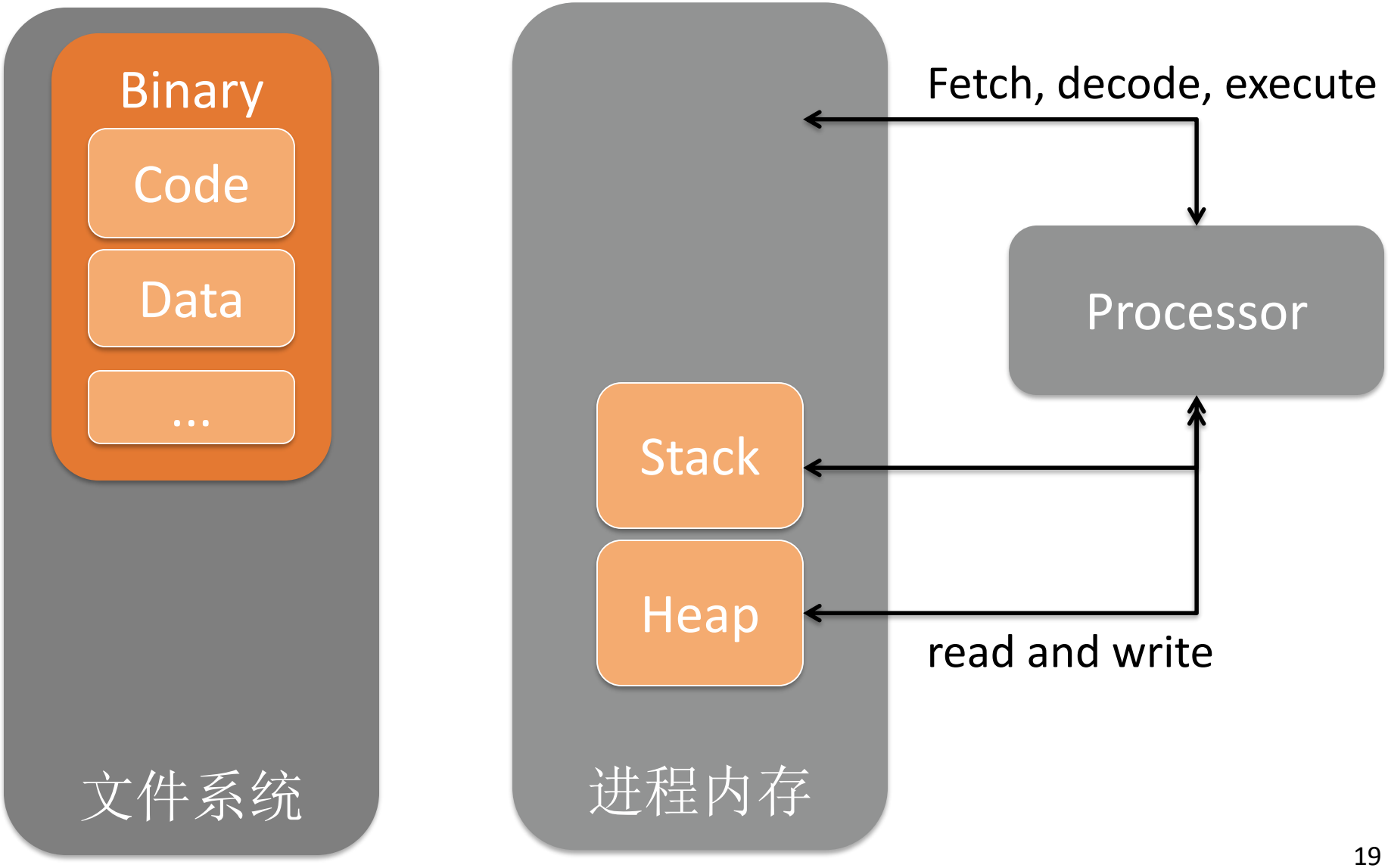

Linux User/Kernel Memory Split
Kernel Space (1GB)
User Mode Space (3GB)
0xffffffff
0xc0000000
0
Windows, default memory split
Kernel Space (2GB)
User Mode Space (2GB)
0xffffffff
0x80000000
0
Windows booted with /3GB switch
Kernel Space (1GB)
User Mode Space (3GB)
Only applies to EXEs flagged as large-address aware.
Kernel Space (1GB)
User Mode Space (Firefox)
Process Switch
Kernel Space (1GB)
User Mode Space (/bin/ls)
Process Switch
Kernel Space (1GB)
User Mode Space (Firefox)

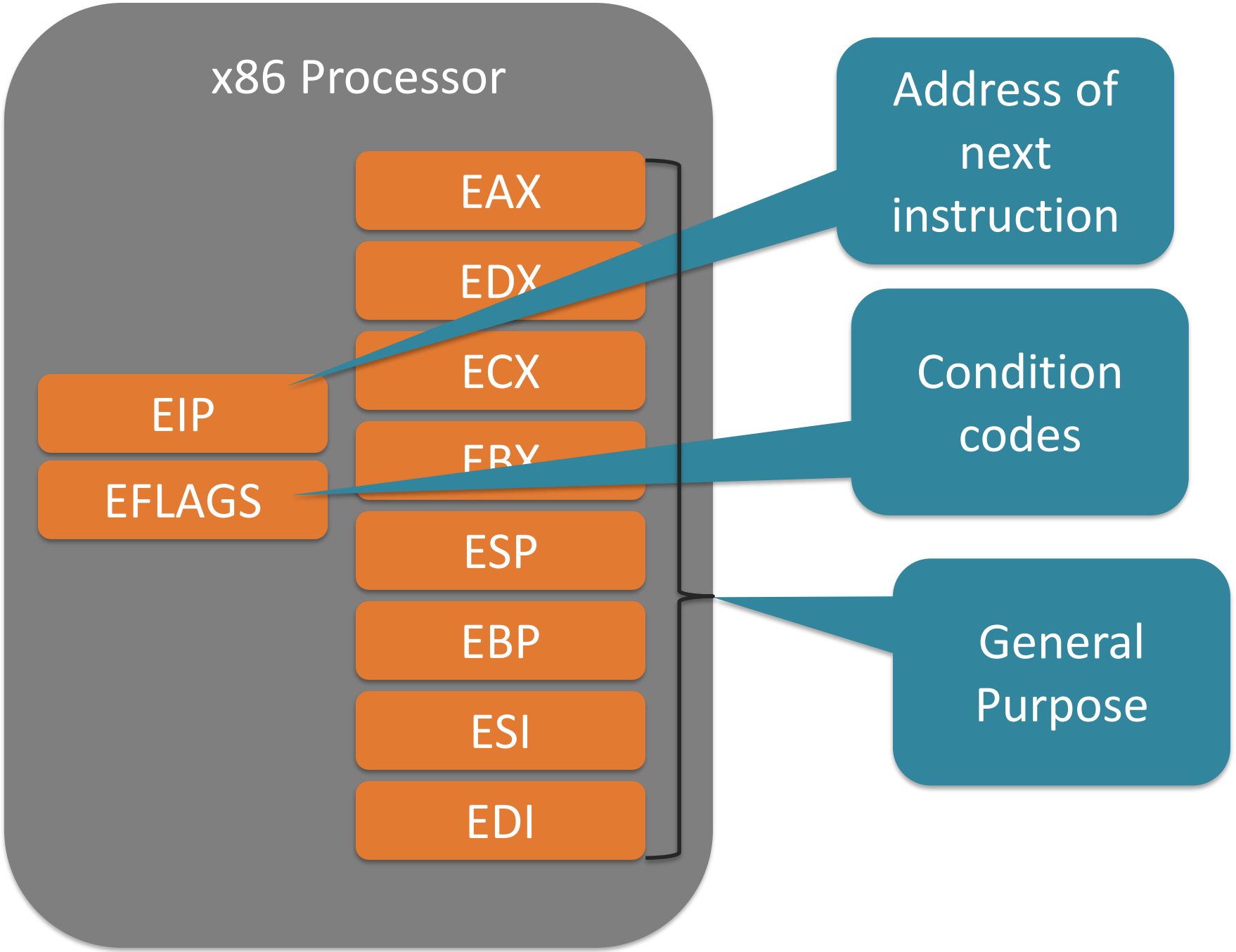
进程切换

内存中的程序

/proc/pid_of_process/maps

基本执行


Binary
Code
Data
...
文件系统
进程内存
Stack
Heap
Processor
Fetch, decode, execute
read and write


x86 Processor
EAX
EDX
ECX
EBX
ESP
EBP
ESI
EDI
EIP
EFLAGS
Address of next instruction
Condition codes
General Purpose

C 和Assembly

C：

- if-then-else
- while
- for loops
- do-while

Assembly：

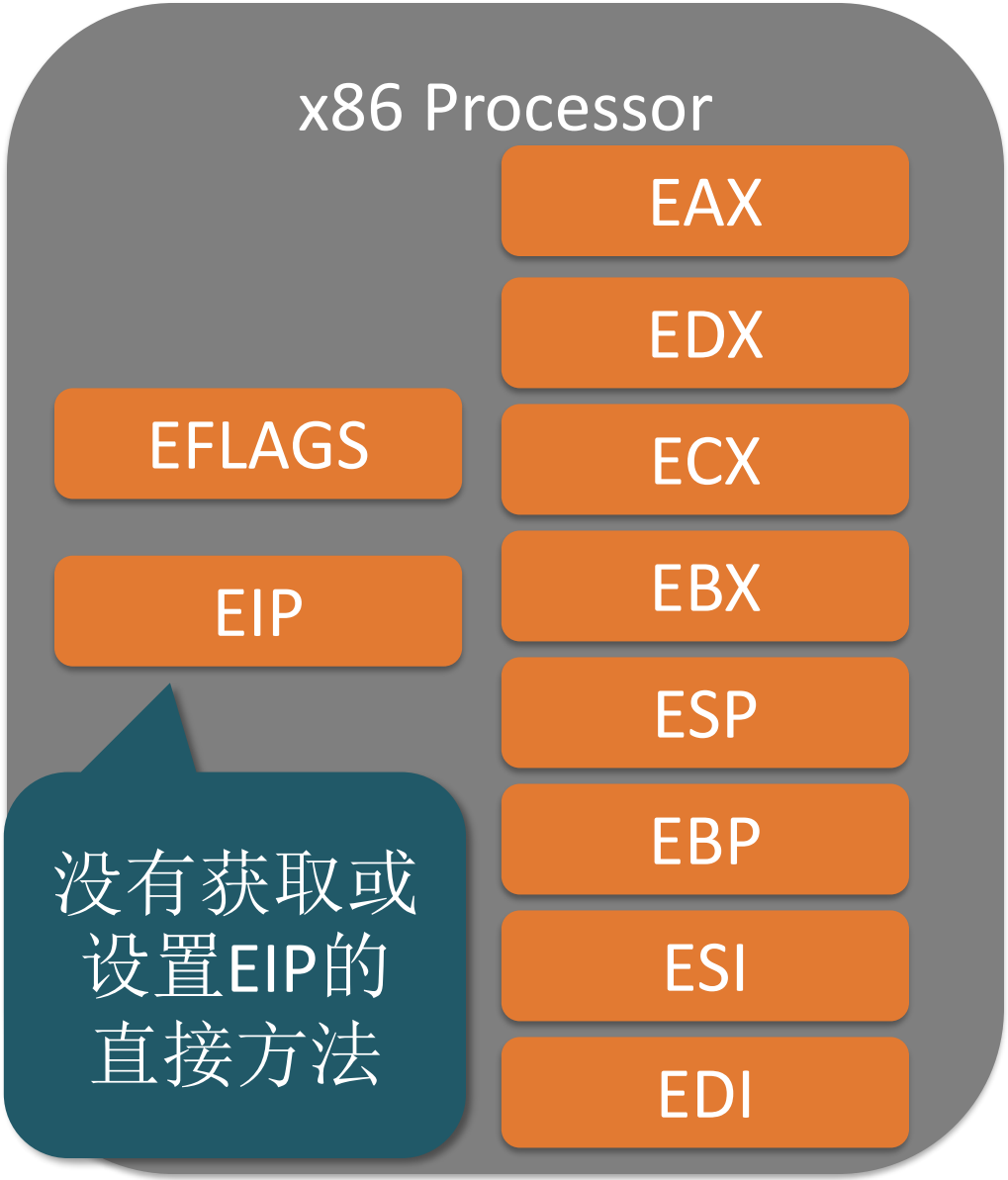
- Jump
- Branch

jumps (跳转)

- **jmp 0x45, called a *direct jump***
- **jmp *%eax, called an *indirect jump***

branches (分支)

- **`if (EFLAG) jmp x`**
  Use one of the 32 EFLAG bits to determine if jump taken



“if”的实现

C

```
1. if(x <= y)
2.    z = x;
3. else
4.    z = y;
```

用两条指令实现：

1. Set eflag to conditional
2. Test eflag and branch

Pseudo-Assembly

A. Test if x <= y by computing x – y. Set EFLAGS.

a. CF set if x < y

b. ZF set if x==y

B. test EFLAGS. If both CF and ZF not set, branch to E

***C. Fall-through:* mov x, z**

D. Jump to F

E. mov y, z

F. <end of if-then-else>

if(x <= y)

eax holds x and 0xc(%ebp) holds y

```
cmp 0xc(%ebp), %eax
ja addr
```

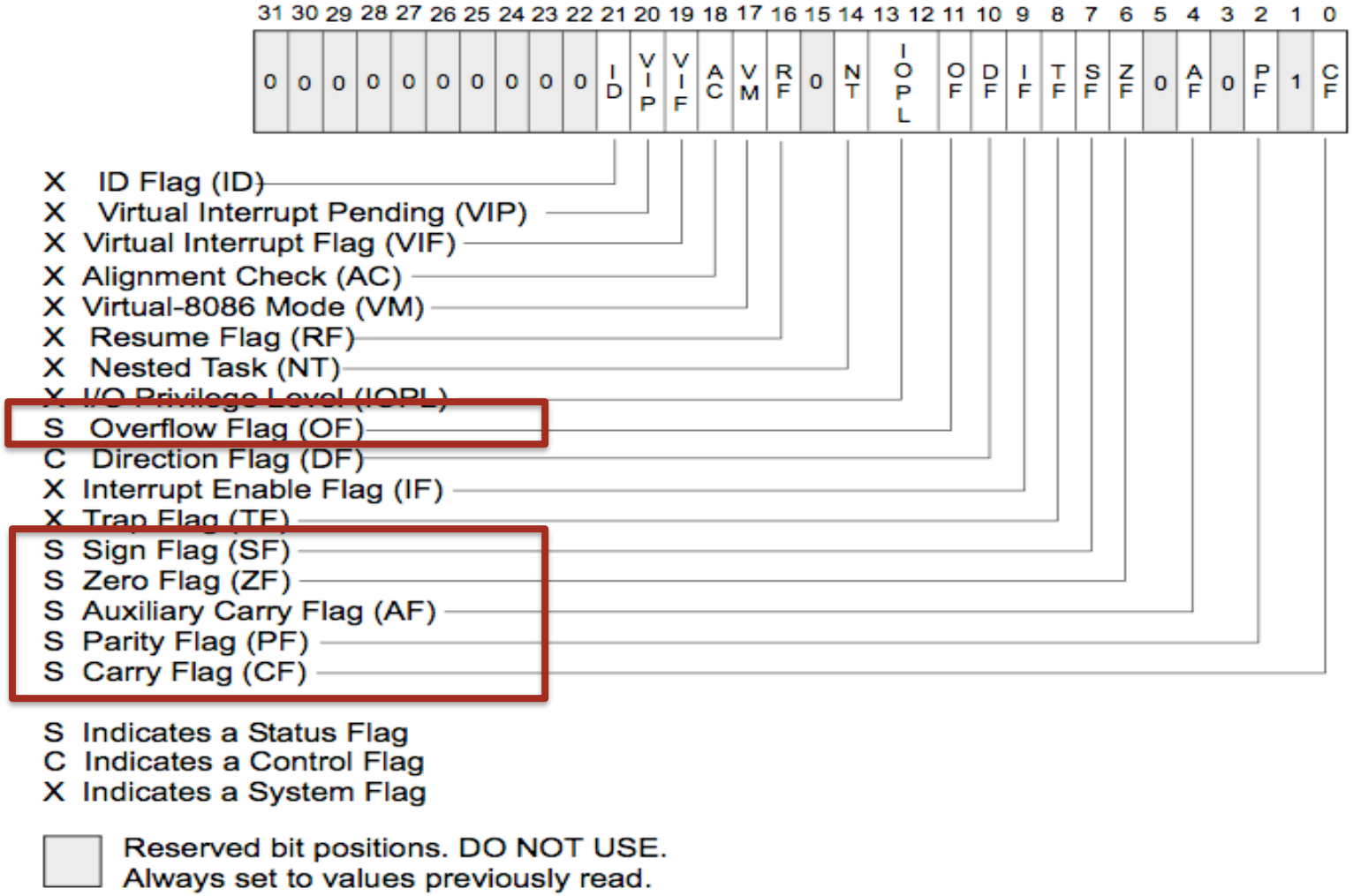
Same as “sub” instruction
r = %eax - M[ebp+0xc], i.e., x – y

Jump if CF=0 and ZF=0

$$(x>=y) \wedge (x!=y) \Rightarrow x > y$$

EFLAGS设置

- **“cmp”和算数运算指令等会设置EFLAGS**
  - CF(bit 0) [Carry flag] 若算术操作产生的结果在最高有效位(most-significant bit)发生进位或借位则将其置1，反之清零
  - ZF(bit 6) [Zero flag] 若结果为0则将其置1，反之清零
  - PF(bit 2) [Parity flag] 如果结果的最低有效字节(least-significant byte)包含偶数个1位则该位置1，否则清零
  - OF(bit 11) [Overflow flag] 如果整型结果是较大的正数或较小的负数，并且无法匹配目的操作数时将该位置1，反之清零。这个标志为带符号整型运算指示溢出状态
  - SF(bit 7) [Sign flag] 该标志被设置为有符号整型的最高有效位。(0指示结果为正，反之则为负)



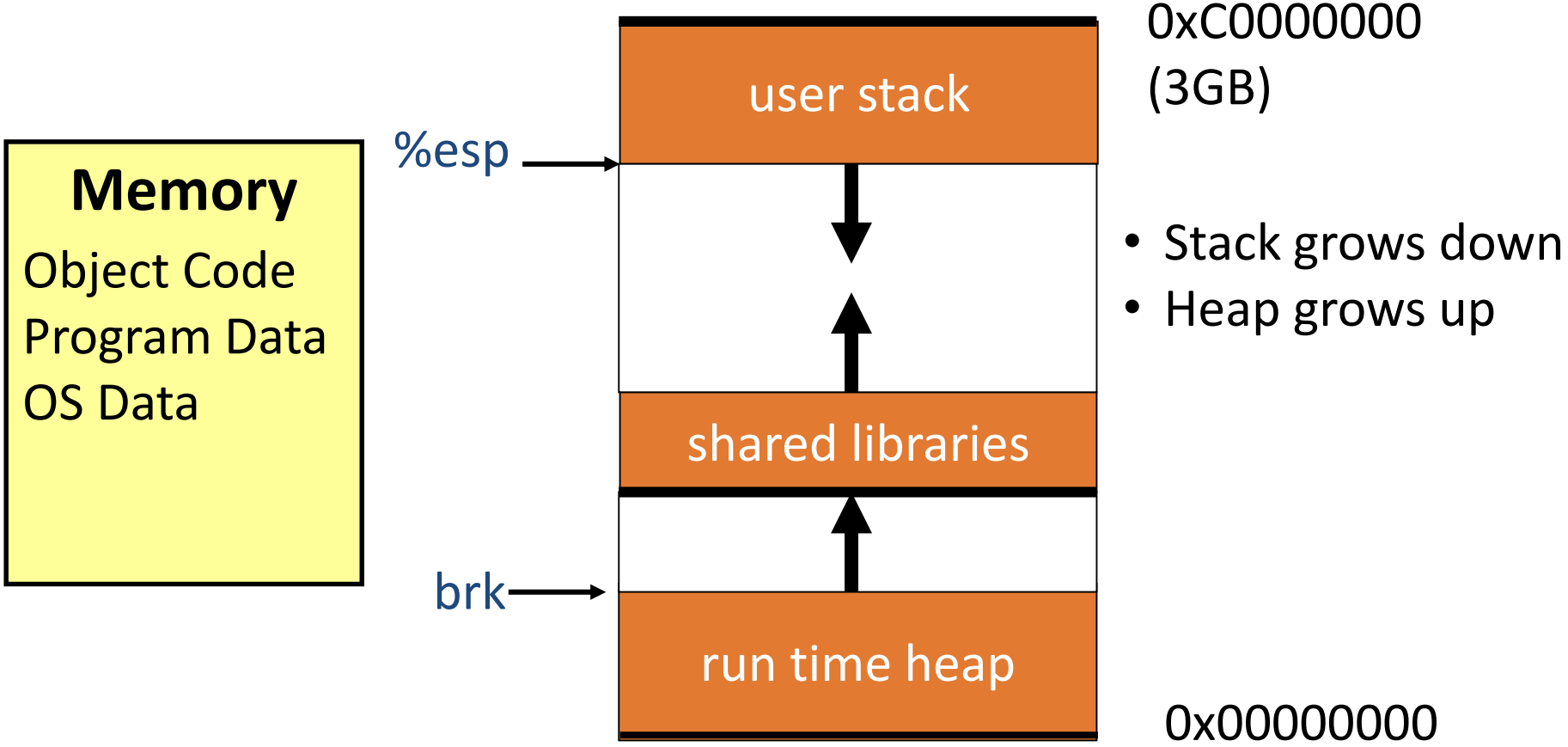
From the Intel x86 manual

跳转指令和EFLAGS

Instr.	Description	Condition
JA	Jump if bigger than	CF==0 and ZF==0
JO	Jump if overflow	OF == 1
JNO	Jump if not overflow	OF == 0
JS	Jump if sign	SF == 1
JZ	Jump if zero	ZF == 1
JE	Jump if equal	ZF == 1
JL	Jump if less than	SF <> OF
JLE	Jump if less than or equal	ZF ==1 or SF <> OF
JB	Jump if below	CF == 1
JP	Jump if parity	PF == 1

C程序控制流(control flow)由以下实现：

- A test on operands
- A branch to a location if not true
- A fall-through to the next instruction if true



栈由高向低生长；堆由低向高生长

变量(Variables)

- 栈
  - 局部变量
- 堆
  - 通过 new/malloc等动态分配.

过程/函数

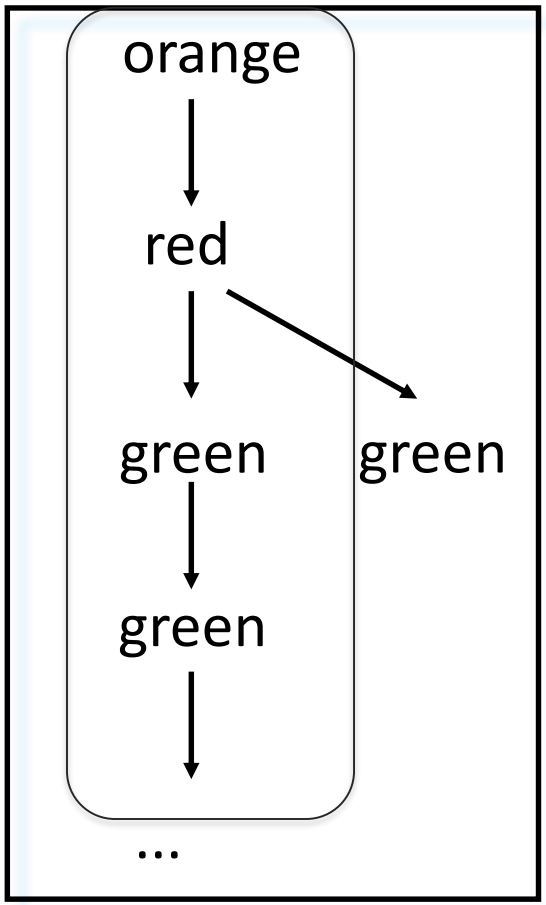
- 需要解决的问题
  - 如何为局部变量分配空间
  - 如何传递参数
  - 如何传递返回值
  - 如何共享8个寄存器
- 方法：栈帧(stack frame)
  - 每个过程调用(procedure invocation)具有自己的栈帧
  - LIFO
    - 如procedure A 调用 B, B’s 帧必须在A的帧前面退出

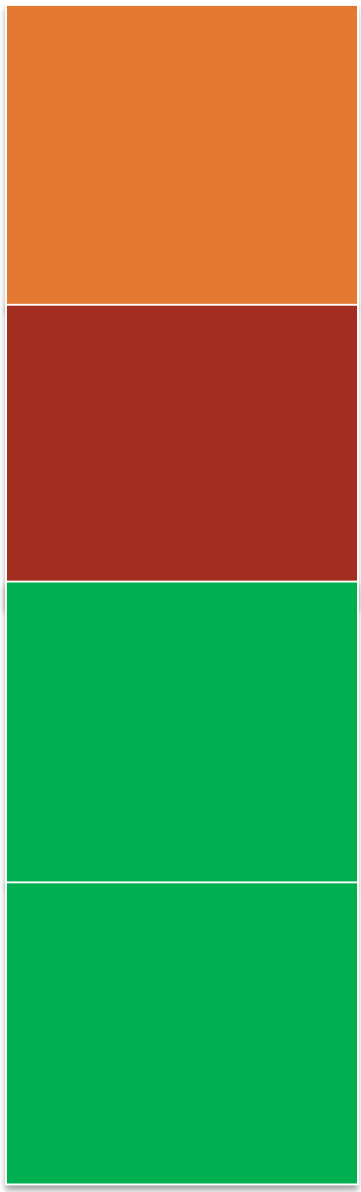
```
orange(…)
{
    …
    red()
    …
}
```

```
red(…)
{
    …
    green()
    …
    green()
}
```

```
green(…)
{
    …
    green()
    …
}
```

函数的调用链



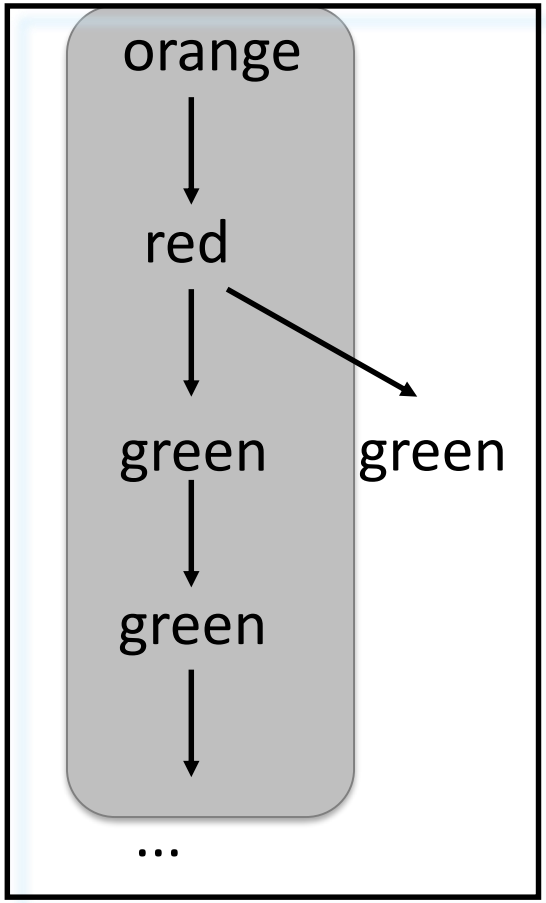


Frame for
- locals
- pushing parameters
- temporary space

Call to red
"***pushes***"
new frame

When green
returns it
"***pops***"
its frame

函数的调用链



On the Stack

```
int orange(int a, int b)
{
   char buf[16];
   int c, d;
   if(a > b)
      c = a;
   else
      c = b;
   d = red(c, buf);
   return d;
```

需要访问参数

需要空间存放局部变量
(buf, c, and d)

需要空间放置被调用函数
(callee)的参数

被调用函数需要途径传回
返回值

调用约定(Calling convention)决定上述特性

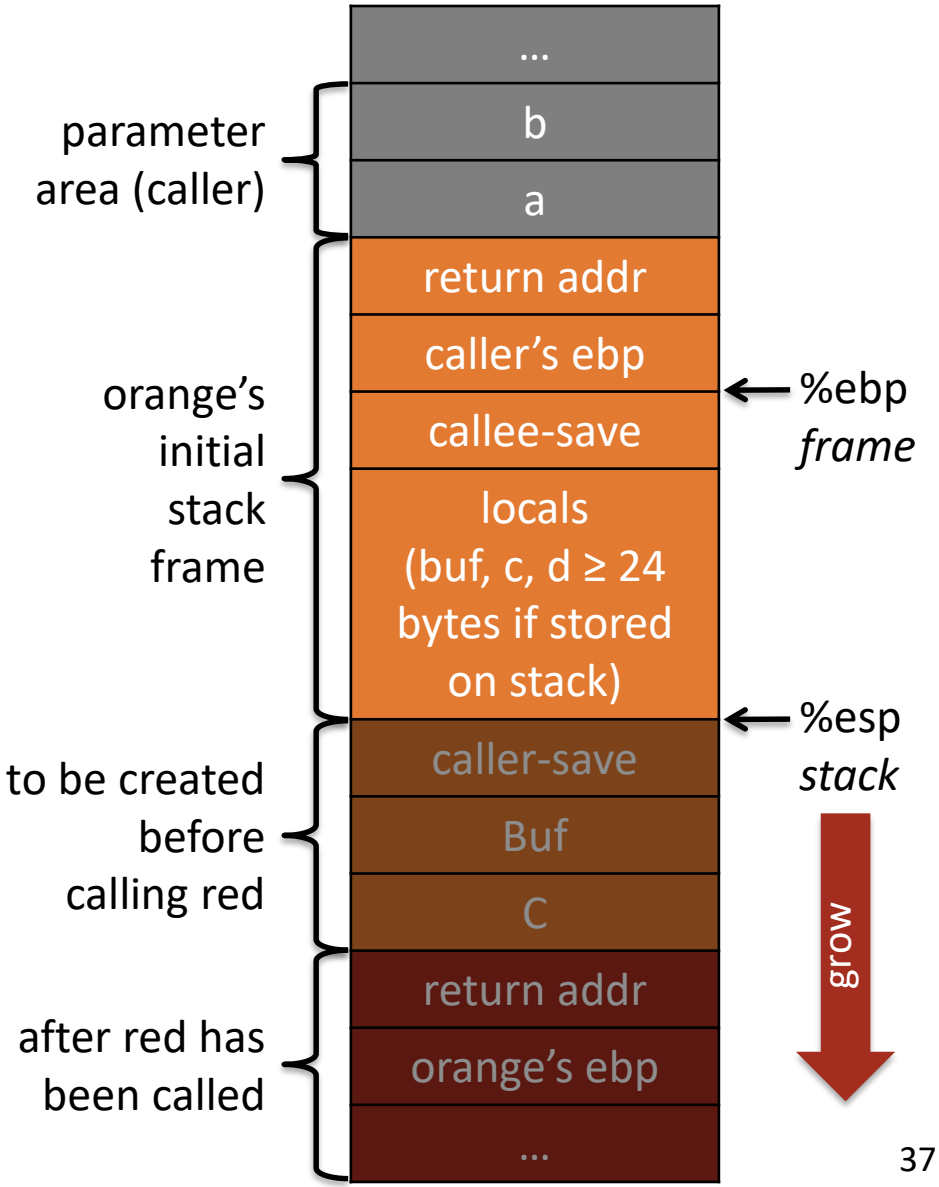
调用约定

- 参数压栈顺序
- 函数命名规则
- 堆栈清理约定
- …

https://en.wikipedia.org/wiki/X86_calling_conventions

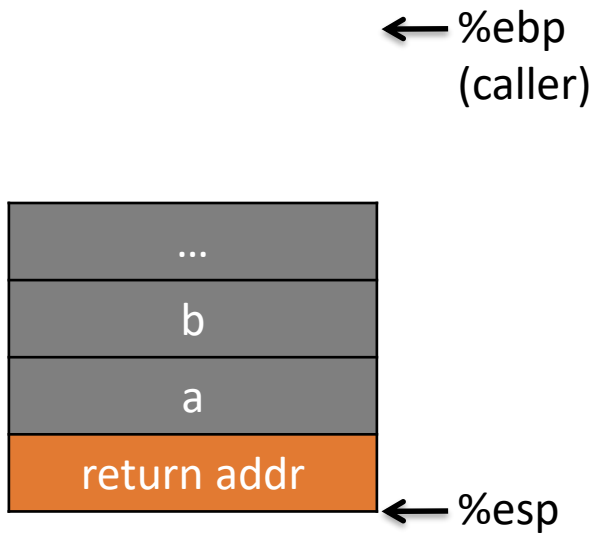
cdecl – Linux & gcc的缺省调用方法

```
int orange(int a, int b)
{
   char buf[16];
   int c, d;
   if(a > b)
      c = a;
   else
      c = b;
   d = red(c, buf);
   return d;
}
```



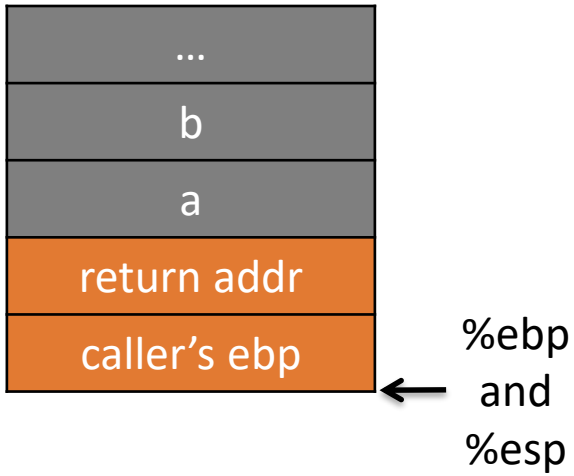
When orange attains control,

1. **return address has already been pushed onto stack by caller**



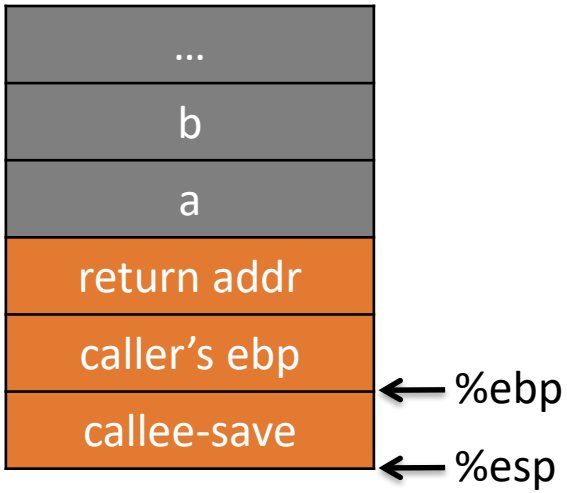
When orange attains control,

1. **return address has already been pushed onto stack by caller**
2. **own the frame pointer**
   - push caller's ebp
   - copy current esp into ebp
   - first argument is at ebp+8



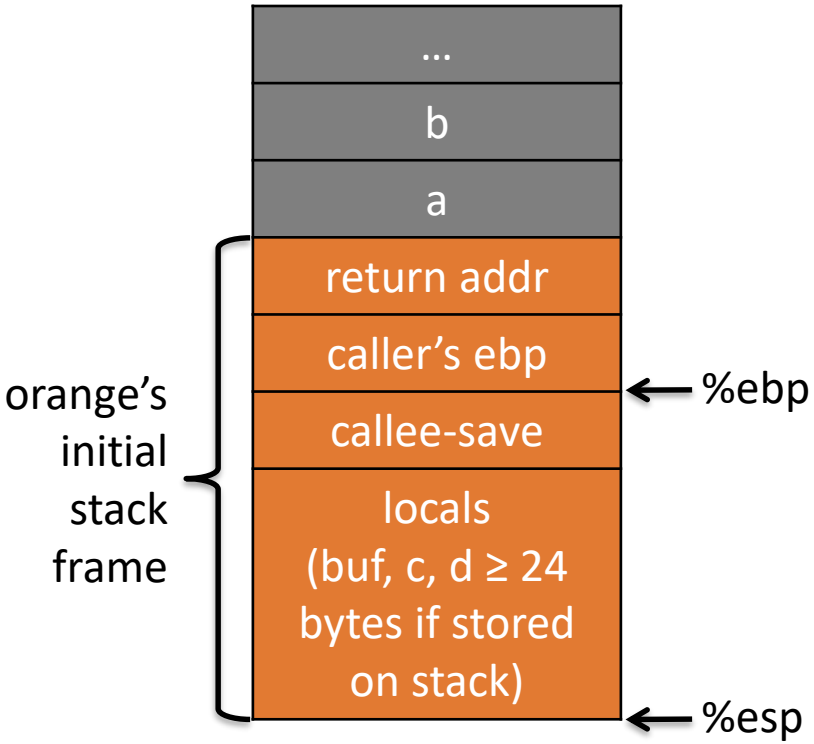
When orange attains control,

1. **return address has already been pushed onto stack by caller**
2. **own the frame pointer**
   - push caller's ebp
   - copy current esp into ebp
   - first argument is at ebp+8
3. **s**ave values of **other callee-save** registers ***if used***
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic

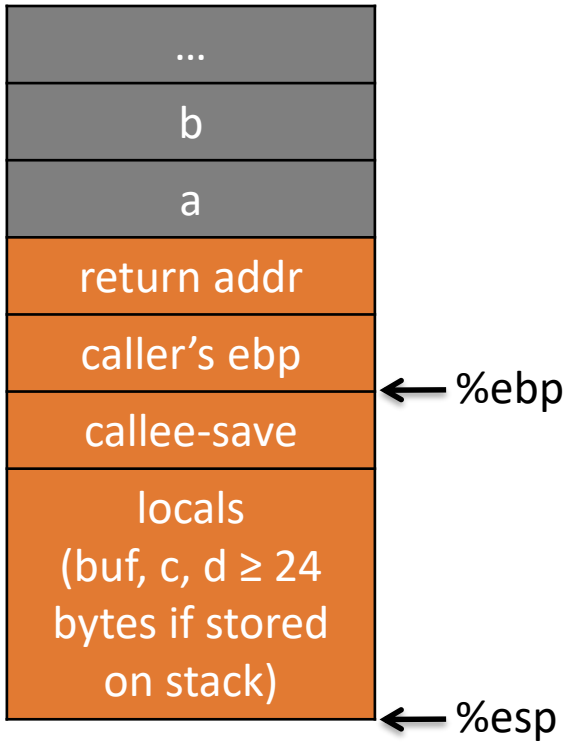


When orange attains control,

1. **return address has already been pushed onto stack by caller**
2. **own the frame pointer**
   - push caller's ebp
   - copy current esp into ebp
   - first argument is at ebp+8
3. **s**ave values of **other** callee-save registers ***if used***
   - edi, esi, ebx: via push or mov
   - esp: can restore by arithmetic
4. **allocate** space for locals
   - subtracting from esp
   - "live" variables which can be "***spilled***" to stack space

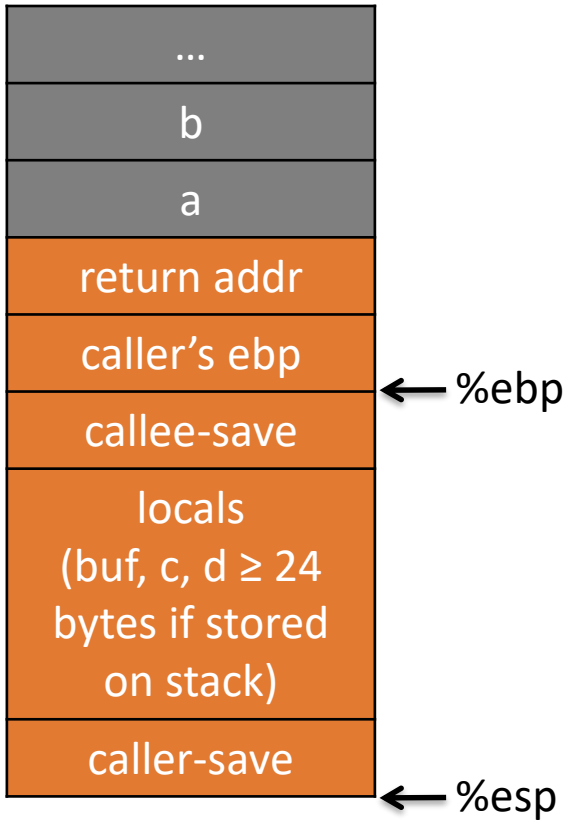


For *caller* orange to call *callee* red,



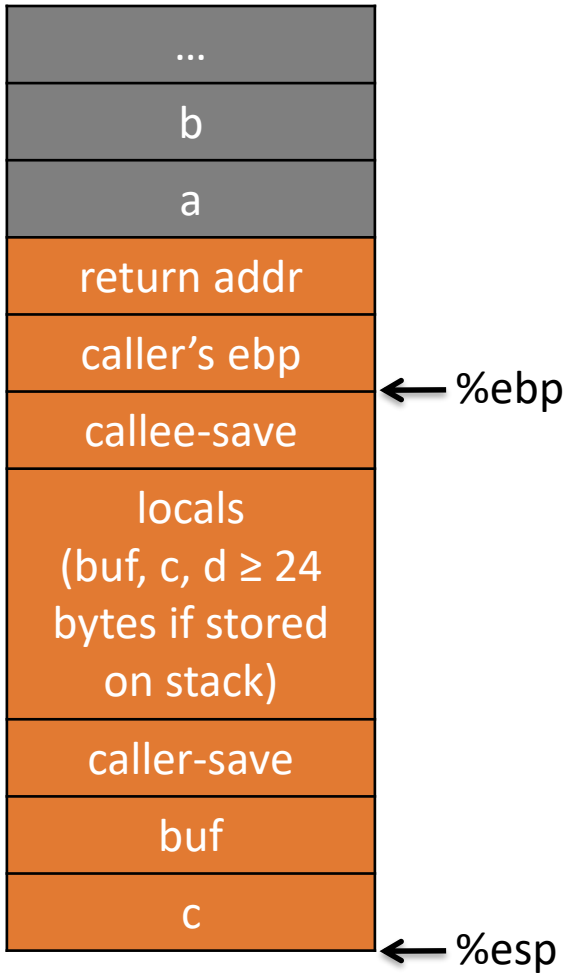
For *caller* orange to call *callee* red,

1. **push any caller-save registers if their values are needed after red returns**
   - eax, edx, ecx



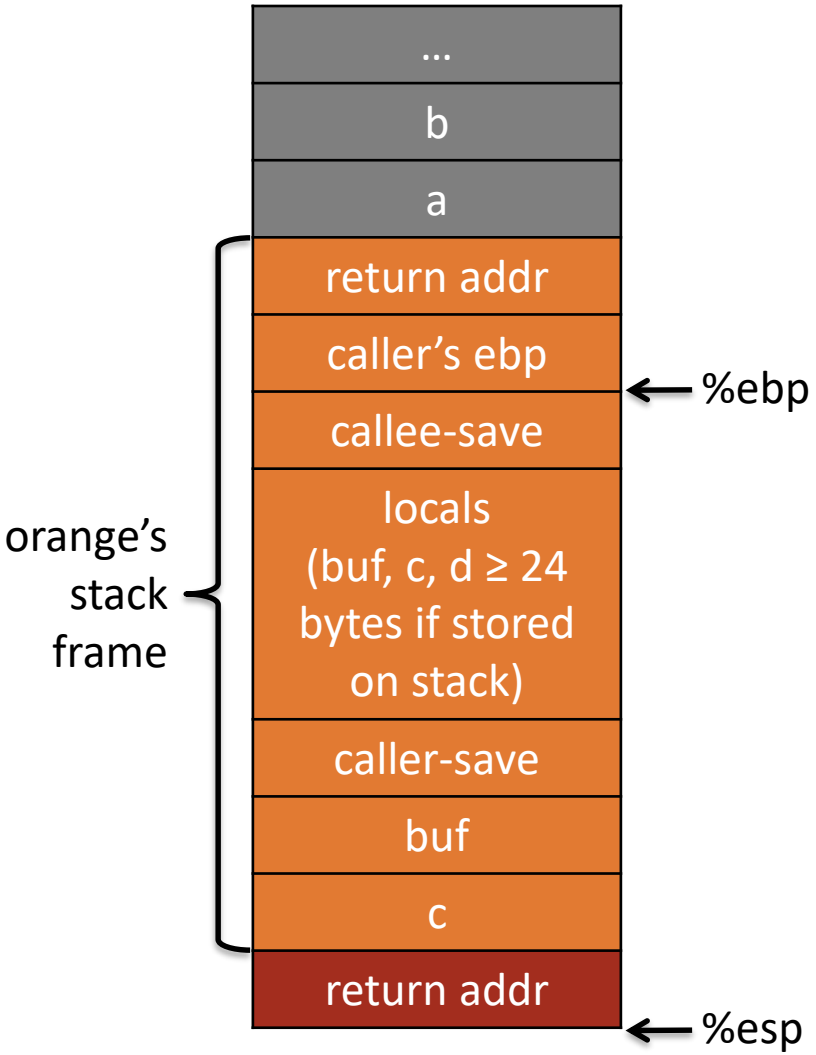
For *caller* orange to call *callee* red,

1. **push any caller-save registers if their values are needed after red returns**
   - eax, edx, ecx
2. **push arguments to red from right to left (reversed)**
   - from callee's perspective, argument 1 is nearest in stack



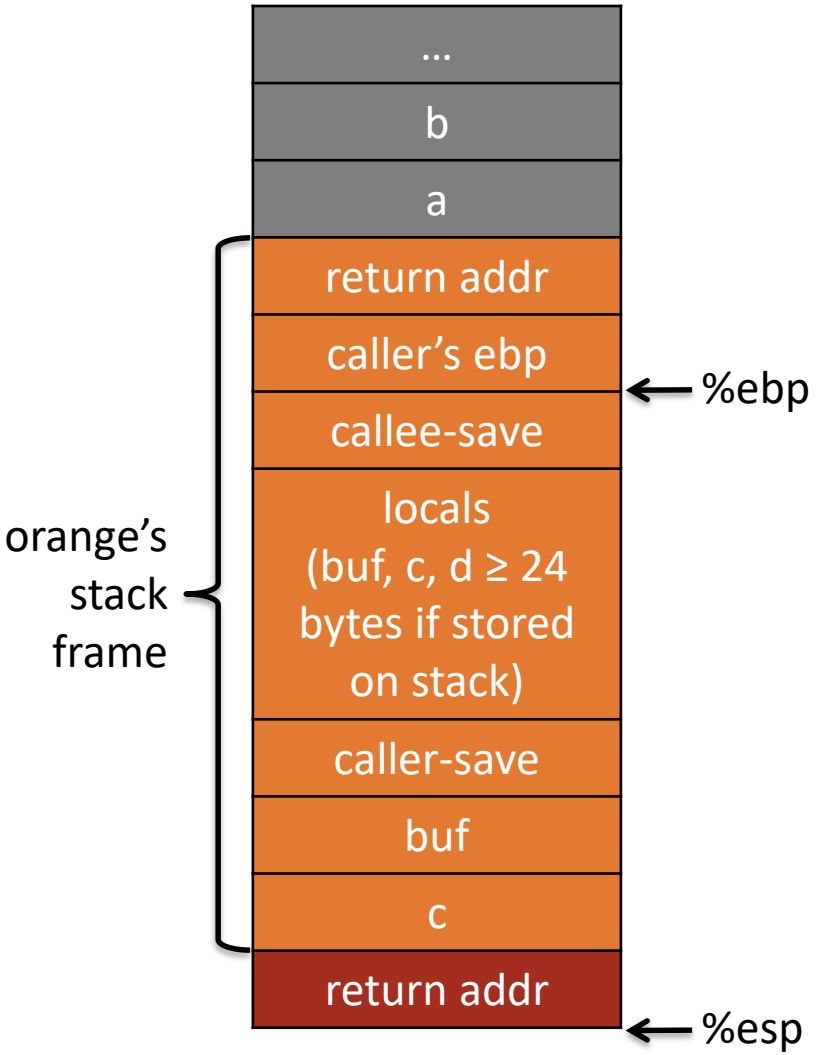
For *caller* orange to call *callee* red,

1. **push any caller-save registers if their values are needed after red returns**
   - eax, edx, ecx
2. **push arguments to red from right to left (reversed)**
   - from callee's perspective, argument 1 is nearest in stack
3. **push return address, i.e., the *next* instruction to execute in orange after red returns**



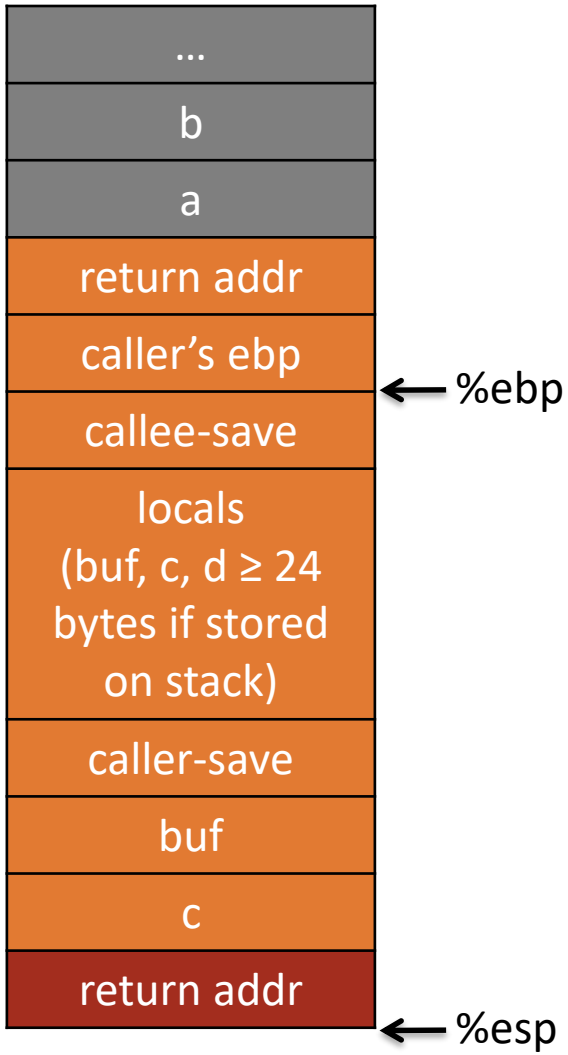
For *caller* orange to call *callee* red,

1. **push any caller-save registers if their values are needed after red returns**
   - eax, edx, ecx
2. **push arguments to red from right to left (reversed)**
   - from callee's perspective, argument 1 is nearest in stack
3. **push return address, i.e., the *next* instruction to execute in orange after red returns**
4. **transfer control to red**
   - usually happens together with step 3 using `call`



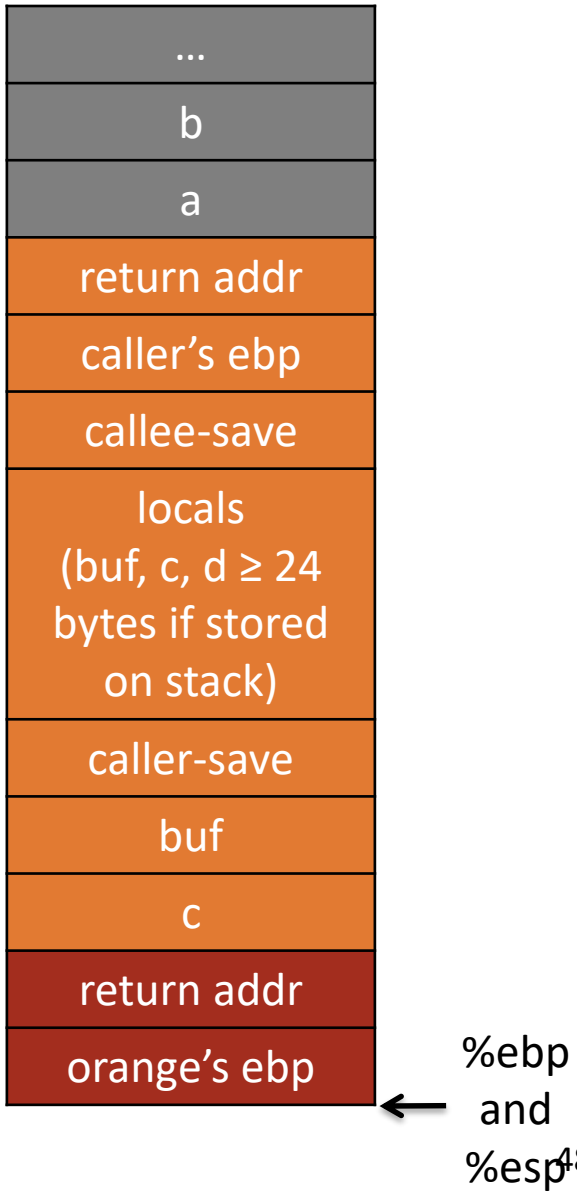
When red attains control,

1. **return address has already been pushed onto stack by orange**



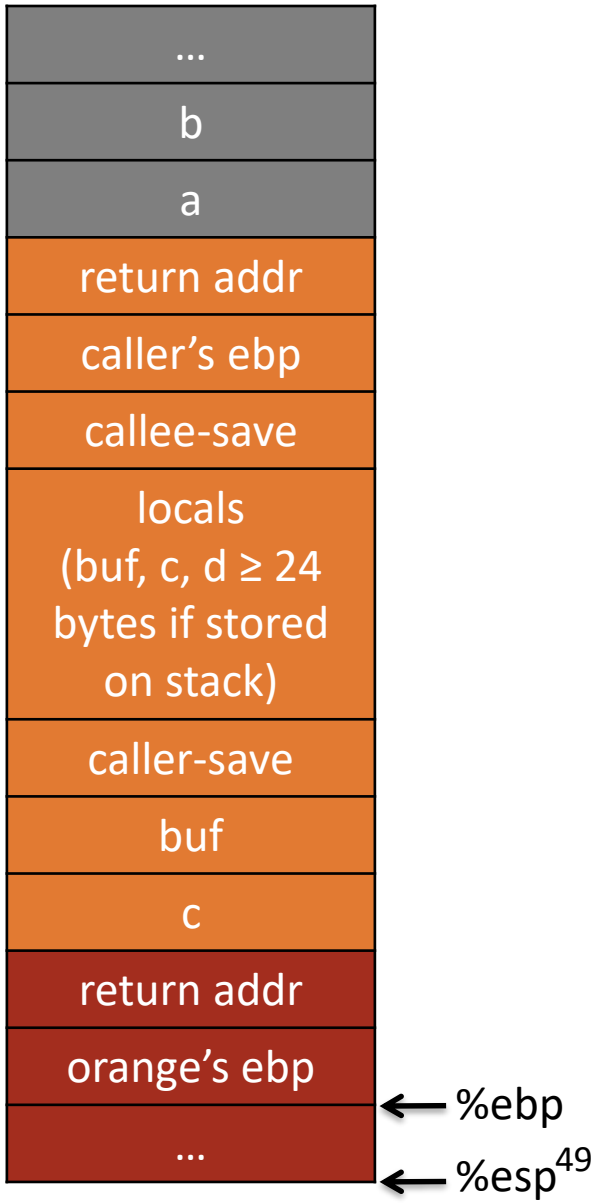
When red attains control,

1. **return address has already been pushed onto stack by orange**
2. **own the frame pointer**



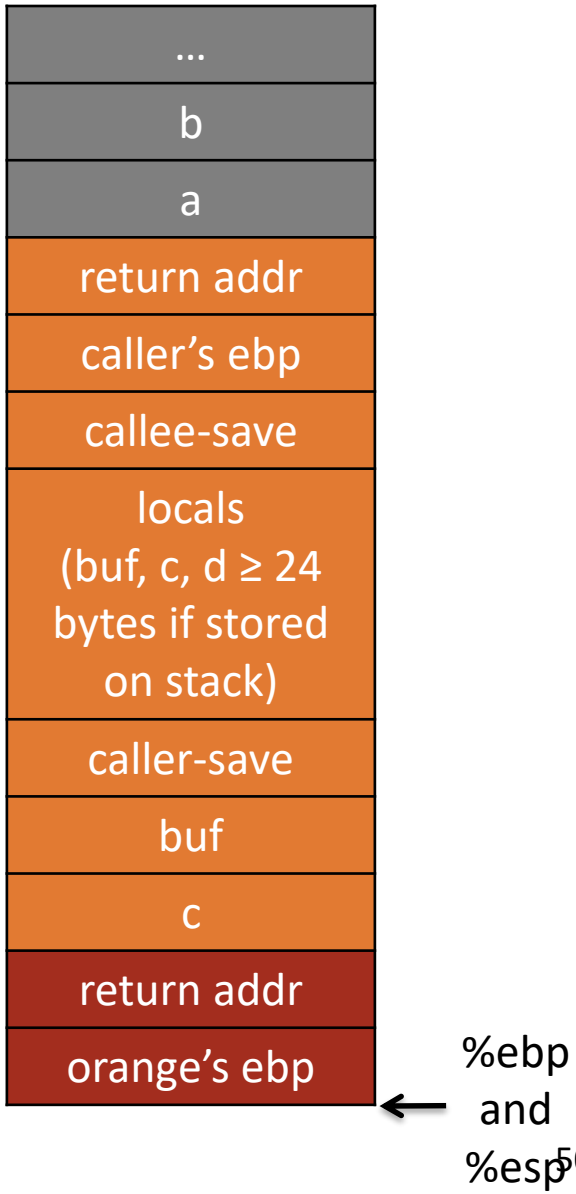
When red attains control,

1. **return address has already been pushed onto stack by orange**
2. **own the frame pointer**
3. **… (red is doing its stuff) …**



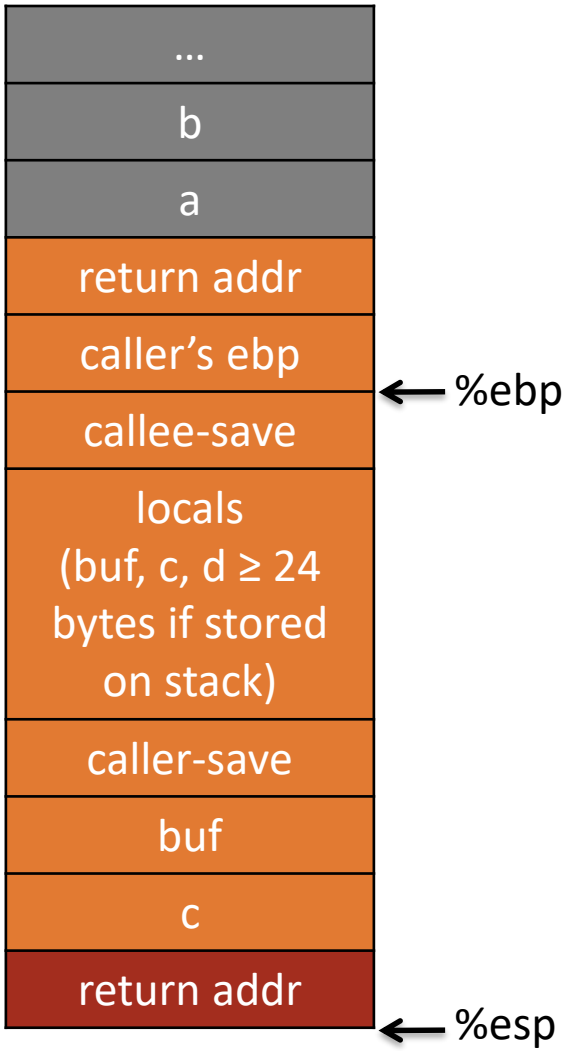
When red attains control,

1. **return address has already been pushed onto stack by orange**
2. **own the frame pointer**
3. **… (red is doing its stuff) …**
4. **store return value, if any, in eax**
5. **deallocate locals**
   - adding to esp
6. **restore any callee-save registers**



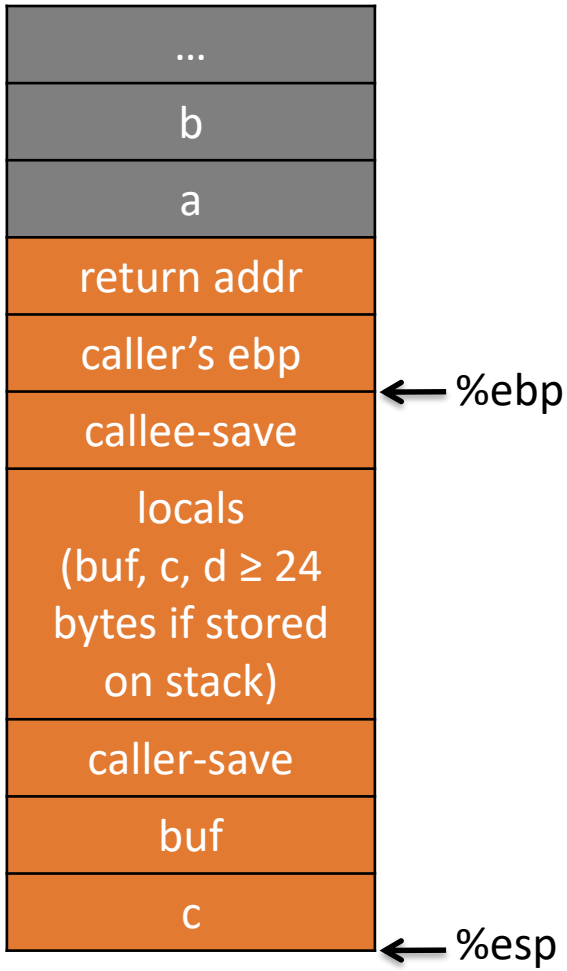
When red attains control,

1. **return address has already been pushed onto stack by orange**
2. **own the frame pointer**
3. **… (red is doing its stuff) …**
4. **store return value, if any, in eax**
5. **deallocate locals**
   - adding to esp
6. **restore any callee-save registers**
7. **restore orange's frame pointer**
   - pop %ebp

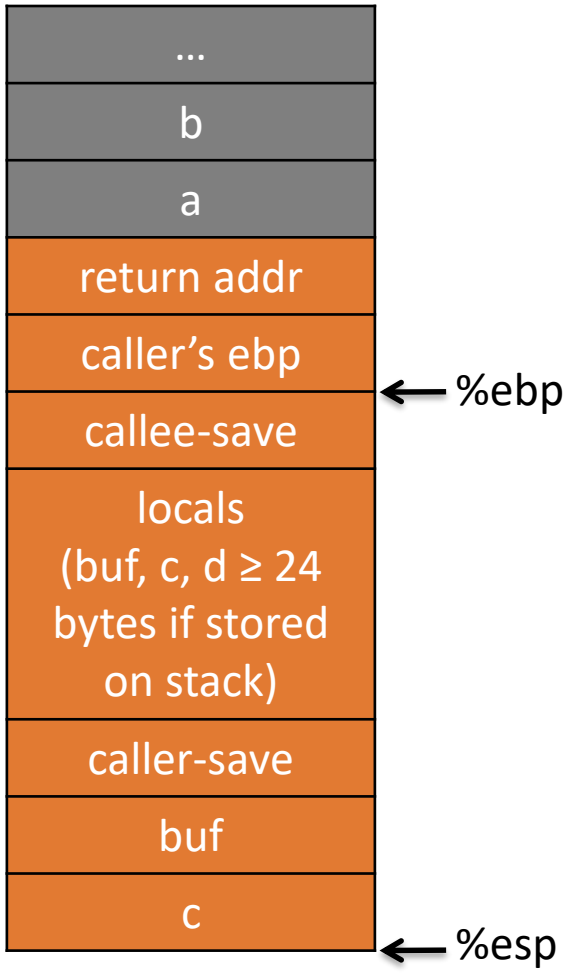


When red attains control,

1. **return address has already been pushed onto stack by orange**
2. **own the frame pointer**
3. **… (red is doing its stuff) …**
4. **store return value, if any, in eax**
5. **deallocate locals**
   - adding to esp
6. **restore any callee-save registers**
7. **restore orange's frame pointer**
   - pop %ebp
8. **return control to orange**
   - `ret`
   - pops return address from stack and jumps there

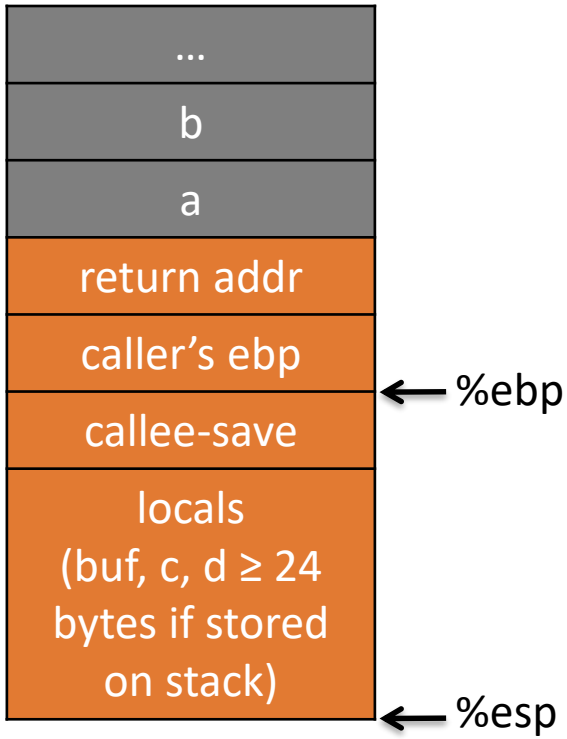


When orange regains control,



When orange regains control,

1. **clean up arguments to red**
   - adding to esp
2. **restore any caller-save registers**
   - pops
3. **…**



cdecl

Action	Notes
caller saves: eax, edx, ecx	push (old), or mov if esp already adjusted
arguments pushed right-to-left	
linkage data starts new frame	call pushes return addr
callee saves: ebx, esi, edi, ebp, esp	ebp often used to deref args and local vars
return value	pass back using eax
argument cleanup	caller's responsibility

cdecl	Unix-like (GCC)		RTL (C)	Caller	When returning struct/class, the calling code allocates space and passes a pointer to this space via a hidden parameter on the stack. The called function writes the return value to this address. Stack aligned on 16-byte boundary due to a bug.

https://en.wikipedia.org/wiki/X86_calling_conventions

术语

- *Function Prologue*(函数序言) - instructions to set up stack space and save *callee* saved registers
- *Function Epilogue*(函数结语) - instructions to clean up stack space and restore *callee* saved registers