微机原理与接口技术

姓名：陈致蓬

单位：中南大学自动化学院

电话： 15200328617

Email ： ZP.Chen@csu.edu.cn

Homepage:

https://www.scholarmate.com/psnweb/homepage

QQ ： 315566683

第 6 章　内存（二）堆栈原理

6.1　堆栈概述

6.2　堆栈工作原理

6.3　堆栈缓冲区溢出

6.1 堆栈概述
6.1.1 堆栈的定义

单片机应用中，堆栈是个特殊存储区，堆栈属于 RAM 空间的一部分，堆栈用于函数调用、中断切换时保存和恢复现场数据。

堆栈特性：先进后出（FILO—First-In/Last-Out)

堆栈中两个重要操作：

PUSH （入栈）：堆栈指针（ SP ）加 1 ，然后在堆栈的顶部加入一 个元素；

POP （出栈）：先将 SP 所指示的内部 RAM 单元中内容送入直接地址寻址的单元中（目的位置），然后再将堆栈指针（ SP ）减 1 。

6.1.2 设计堆栈的目的

设计目的：减少程序内存（RAM）的占用。

出于成本考虑，单片机的内存容量非常有限。而在单片机上电后，所有的变量又都需要被拷贝到内存中运行。为了解决这个矛盾，尽可能节约代码在运行过程中变量所占用的内存空间，“堆栈”和“局部变量”这两个概念就此诞生。

1、局部变量：从软件角度指出了某些变量只需要在特定的时间段【生存期】存在于单片机的内存中即可满足程序正确运行的要求；

2、堆栈：从硬件角度为程序员控制局部变量的生存期提供了便利。

6.1.3 堆栈功能的实现—如何减少内存的占用

软件角度—局部变量：

某些变量只被某个函数调用（即局部变量），对于这种变量，在函数调用时声明，调用完后就将其内存空间释放，让给其他函数使用，大大提高了内存的利用率。

6.1.3 堆栈功能的实现—如何减少内存的占用

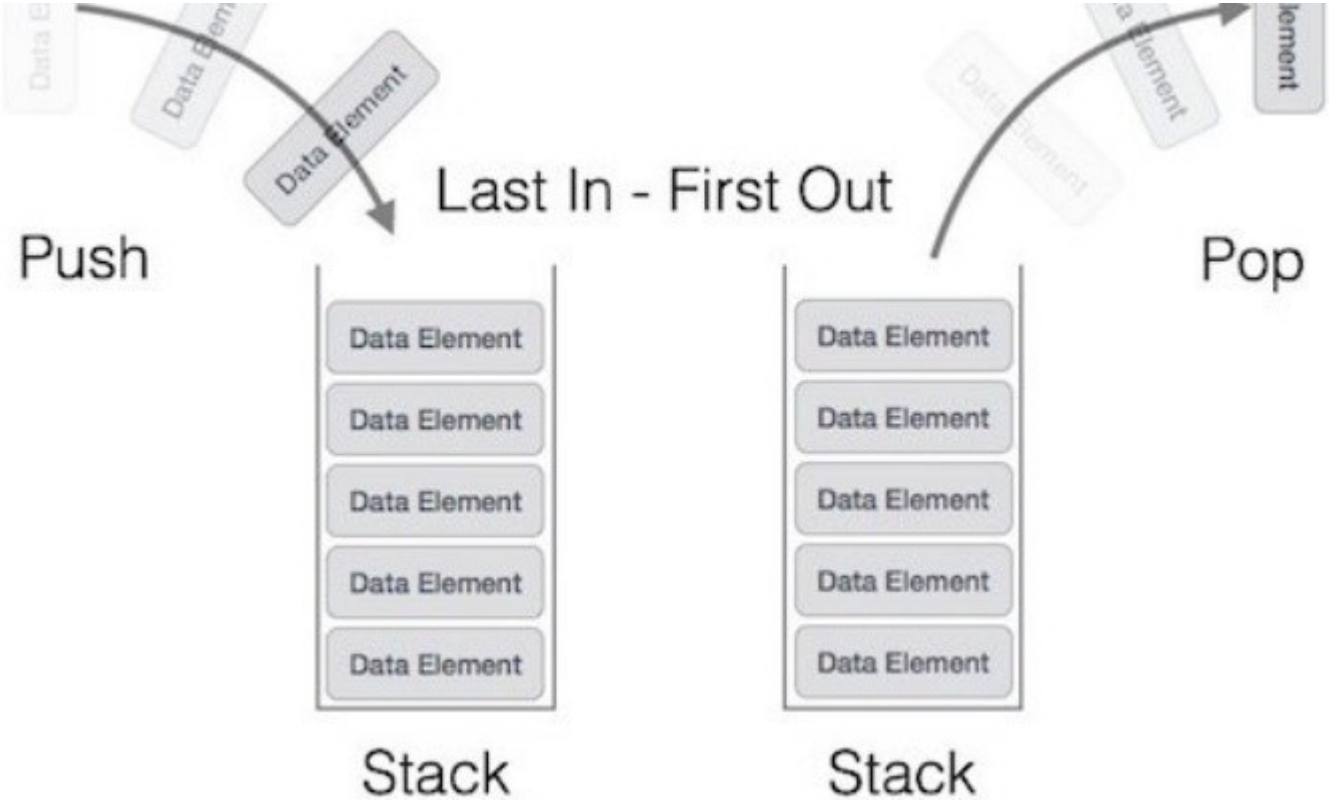
硬件角度—堆栈：

栈并不是一个纯粹抽象的软件概念，而是由包括 esp ， ebp 这些实际存在的堆栈寄存器来支撑的。

- Esp 寄存器：存储栈顶地址
- Ebp 寄存器： 存储栈基地址

Esp 到 ebp 地址之间被视为当前函数的栈空间，声明变量时，根据局部变量的大小，将堆栈寄存器 esp 的值下移，即可留出对应的空间用于存储该变量。而删除时，只需要将当前 ebp 的值赋给 esp 【即将栈顶指针直接压倒栈底】，即可自动将所有与该函数相关的局部变量清空，将所占用空间释放。一条指令，统一管理，统一释放。

6.1.3 堆栈功能的实现—如何减少内存的占用



6.1.4 堆栈的优点

1、地址空间连续，没有碎片化，利用率高；

2、有专门的寄存器帮助，管理方便， push ， pop 一键删除；

3、生存周期短，函数调用结束即释放，所以同一 RAM 空间可以被不同函数 反复使用；

4、因为有了局部变量的概念，就基本杜绝了其他函数误操作局部变量的可能，除非使用指针；

5、避免变量重名。

6.2 堆栈工作原理

程序的内部操作主要包括：

1、函数相互调用时将数据压入堆栈；

2、函数相互调用时从堆栈弹出数据；

3、在堆上为必须跨函数调用时需要存活的数据分配内存。

6.2 堆栈工作原理
6.2.1 栈帧

当调用一个函数时，将创建一个**栈桢**来支持该函数的执行。

栈桢包含：

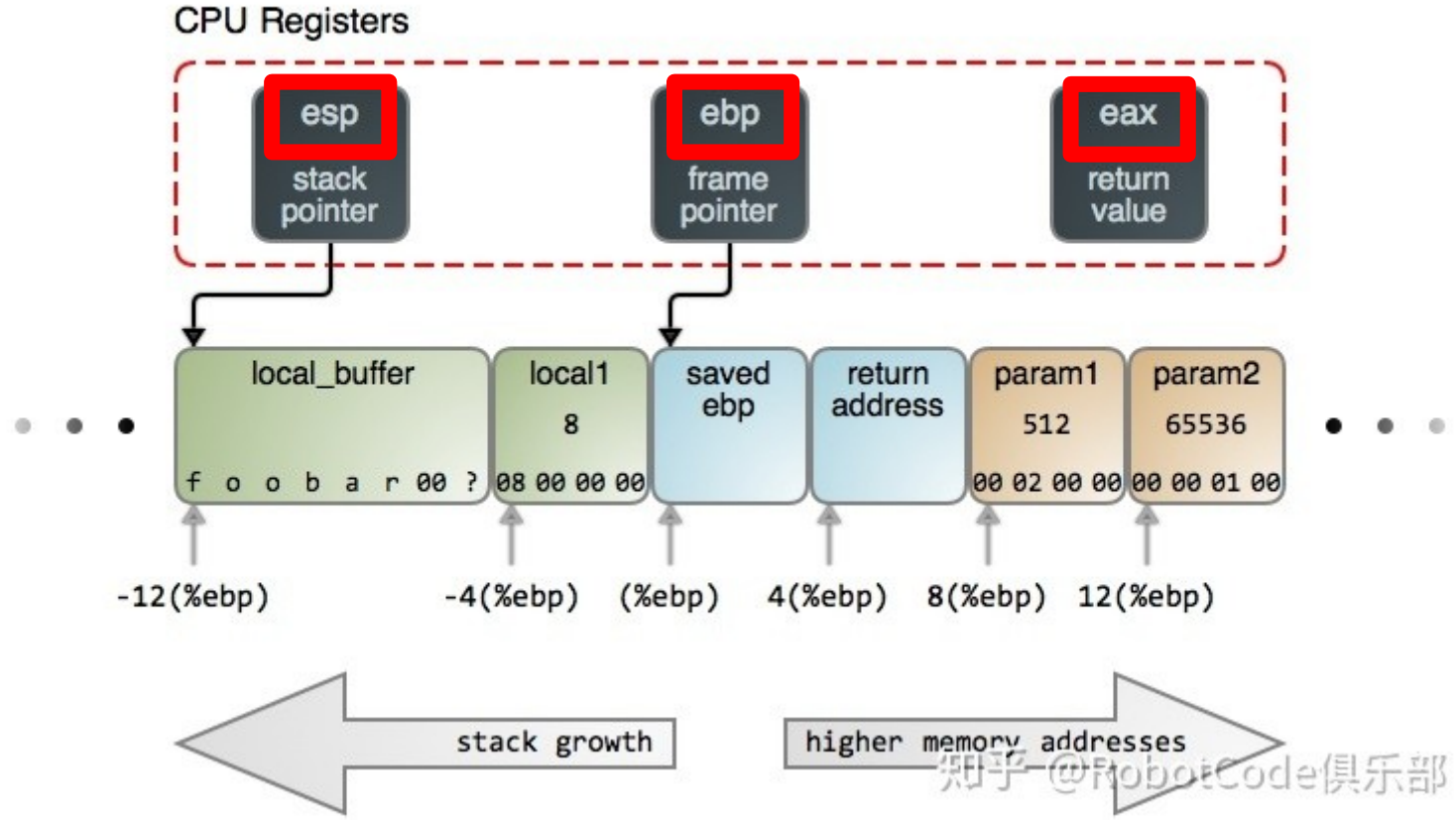
1、函数的**局部变量**；

2、调用者传递给函数的**参数**；

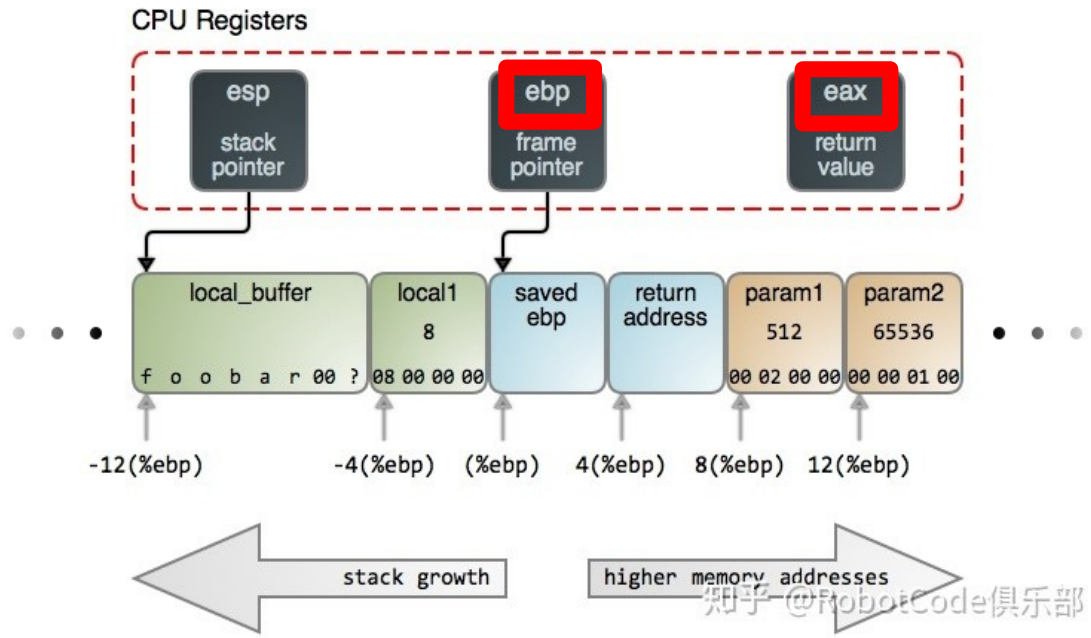
3、**管理信息**，允许被调用的函数（被调用方）安全地返回给调用方。

6.2 堆栈工作原理
6.2.1 栈帧

堆栈的确切内容和布局因处理器体系结构和函数调用约定的不同而不同。如图，以 c 风格函数调用 (cdecl) 的 Intel x86 堆栈为例，其有一个位于栈帧位于栈的顶部。

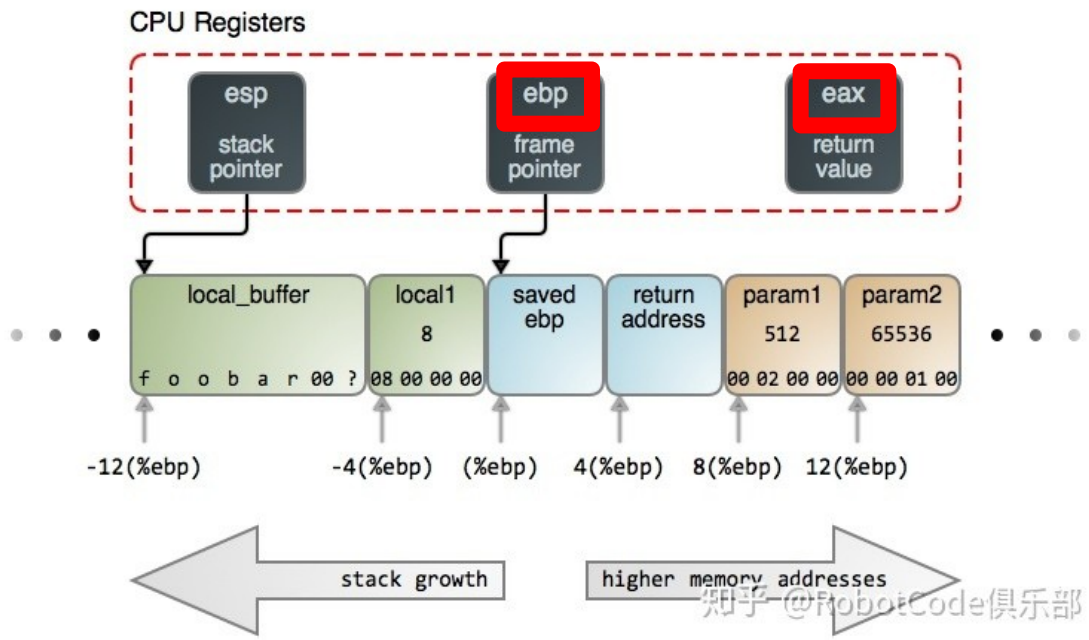


6.2 堆栈工作原理
6.2.1 栈帧



- **ESP** ：指向堆栈的顶部。顶部总是被最后一个被压入到堆栈上但尚未弹出的项目所占据。存储在 esp 中的地址随着栈的内容的压入和弹出而不断变化，因此它总是指向最后一项。许多 CPU 提供了指令自动更新 esp ，没有这个寄存器使用堆栈是不切实际的。在 Intel 架构中，与大多数架构中一样，堆栈的内存地址越来越低，即向地址小的地方增长。因此，“top” 是包含实时数据的堆栈中最低的内存地址，在本例中是 local_buffer 。从 esp 到 local_buffer 的箭头专门指向 local_buffer 占用的第一个字节，这是存储在 esp 中的确切地址。

6.2 堆栈工作原理
6.2.1 栈帧



- **EBP**：基指针（帧指针）。它指向当前运行的函数的栈桢中的一个固定位置，并为访问函数参数和本地变量提供一个稳定的参考点（基）。**ebp 仅在函数调用开始或结束时更改**。因此，我们可以很容易地将堆栈中的每个项作为 ebp 的偏移量来处理，如图所示。与 esp 不同，ebp 主要由程序代码维护，很少受到 CPU 干扰。有时候，完全抛弃 ebp 会带来性能上的好处，这可以通过编译器标志来实现，如 Linux 内核。

- **EAX**：于将大多数 C 数据类型的返回值传输回调用方。

6.2 堆栈工作原理
6.2.1 栈帧

上图显示了在调试器中可以看到的每个字节的精确内容，内存从左到右，从上到下不断增长。这里是：

- **local_buffer** ：一个字符数组，其中包含一个以 null 结尾的 ascii 字符串。字符串可能是从某处读取的，例如键盘输入或文件，它有 7 字节长。因为 local_buffer 可以容纳 8 个字节，所以还剩下 1 个空闲字节。这个字节的内容是未知的，因为在堆栈的无限的压入和弹出之中，无法获知内存中保存的是什么，除非对它进行写入。由于 C 编译器不初始化栈帧的内存，所以内容是不确定的；
- **Local1** ：一个 4 字节的整数，可以看到每个字节的内容。它看起来是个大数，所有的 0 都跟在 8 后面，但是这里你的直觉让你误入歧途。英特尔处理器是小端计算机，内存中的数字从小端开始。因此，多字节数的最小有效字节位于最低内存地址中。因为这通常显示在最左边，这与我们通常的数字表示方式不同。 local1 实际上是 8 ；
- **param1** ：在第二个字节位置的值是 2 ，因此它的数学值是 2 * 256 = 512(乘以 256 ，因为每个位置值的范围是从 0 到 255) ；
- **Param2** ： 1 * 256 * 256 = 65536 。

6.2 堆栈工作原理
6.2.1 栈帧

- 这个栈桢中的管理数据由两个关键部分组成：（上图中中间的浅蓝色部分）

1、 **saved ebp**: 前一个栈桢的地址；

2、 **return address**: 函数退出时要执行的指令的地址。

它们一起使函数能够正常返回，使程序能够继续运行。

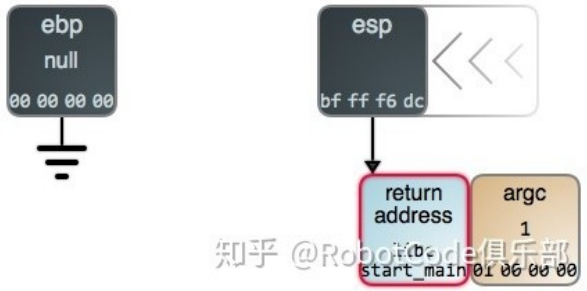
6.2 堆栈工作原理
6.2.2 函数栈帧的构建

```
int add(int a, int b)
{
    int result = a + b;
    return result;
 }
int main(int argc)
 {
    int answer;
    answer = add(40, 2);
 }
```

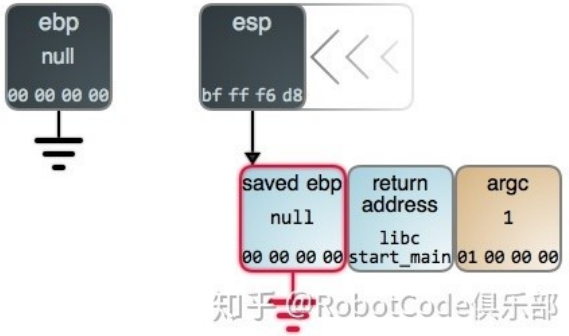
要在栈上分配 8 个字节，就要从 esp 中减去 8 。假设我们在没有命令行参数的 Linux 中运行该程序。当你运行一个 C 程序时，第一个实际执行的代码是在 C runtime 库中，然后它调用我们的 main 函数。

6.2 堆栈工作原理
6.2.2 函数栈帧的构建

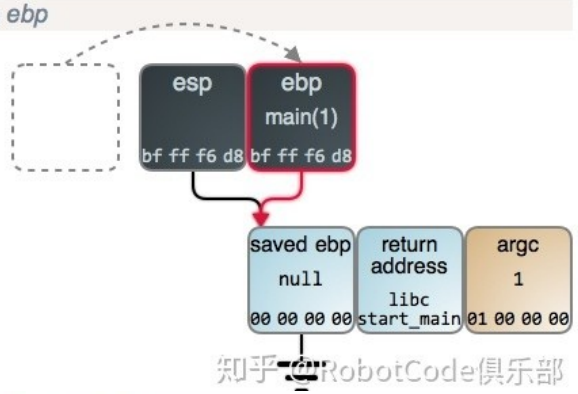
1. `call main` *# push return address onto stack, jump into main*



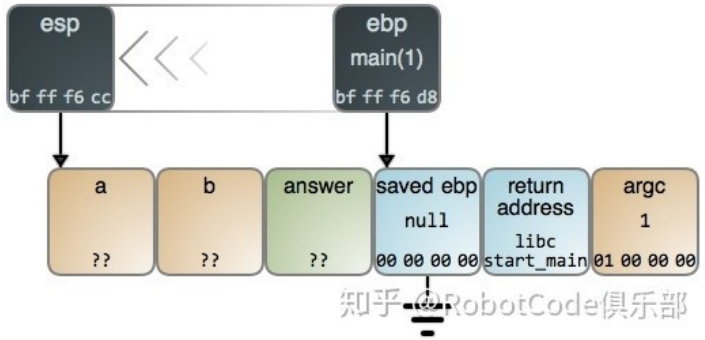
2. `pushl %ebp` *# save current ebp register value*



3. `movl %esp, %ebp` *# copy esp to ebp*



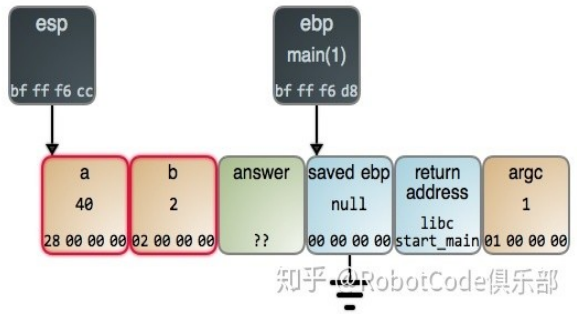
4. `subl $12, %esp` *# make room for stack data*



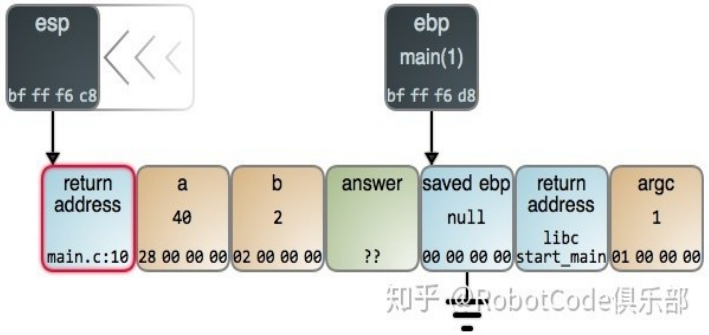
步骤 2 、 3 、 4 是函数序言，这对于几乎所有函数都是通用的：将 ebp 的当前值保存到栈的顶部，然后将 esp 复制到 ebp ，建立一个新的栈桢。 main 的序言与其他任何序言一样，但有一个特性，即在程序启动时 ebp 被归零。

6.2 堆栈工作原理
6.2.2 函数栈帧的构建

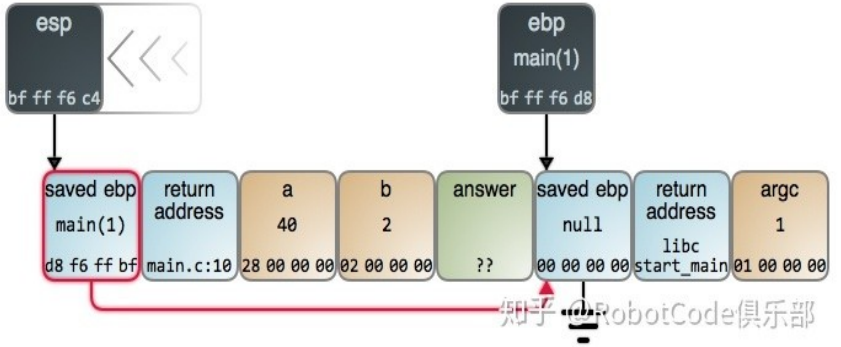
5. `movl $2, 4(%esp) # set b to 2`
`movl $40, (%esp) # set a to 40`



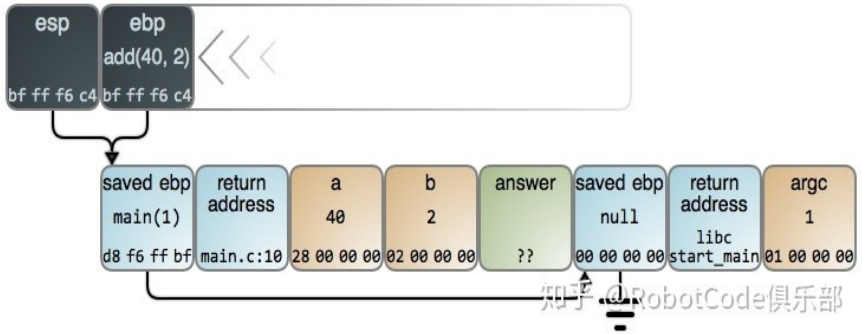
6. `call add # push return address onto stack, jump into add`



7. `pushl %ebp # save current ebp register value`



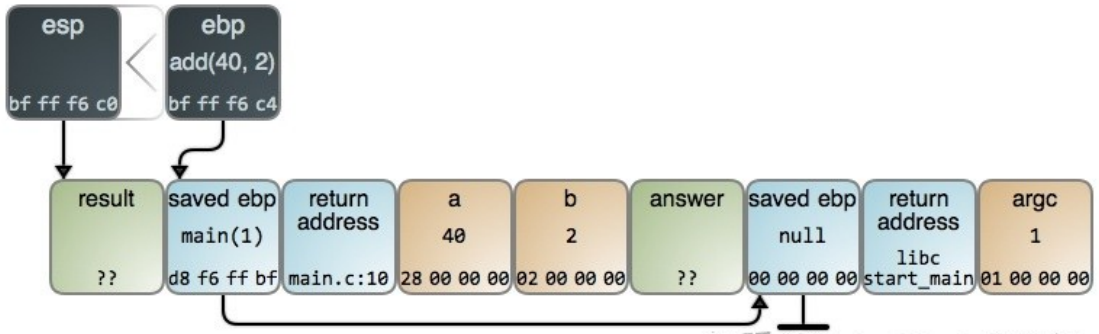
8. `movl %esp, %ebp # copy esp to ebp`



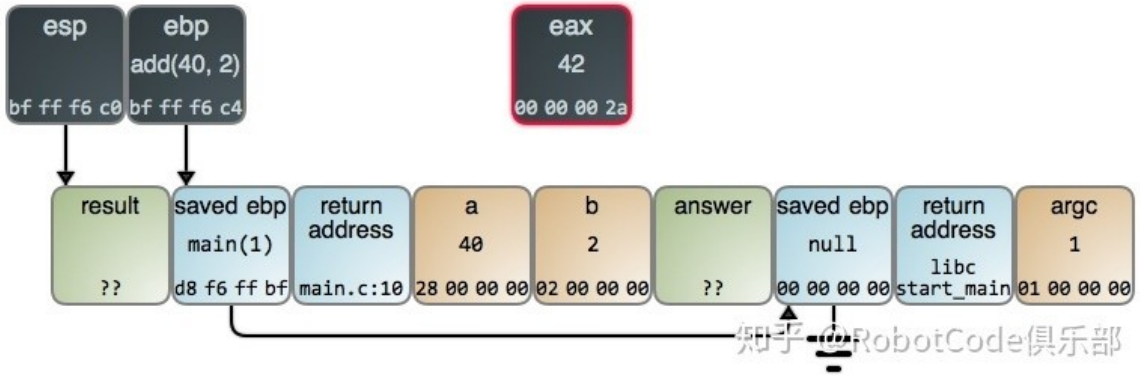
在 main 从 esp 中减去 12 以获得所需的堆栈空间之后，它为 a 和 b 设置值。内存中的值以十六进制和 little-endian 格式显示，就像在调试器中看到的那样。一旦设置好参数值， main 调用 add 并开始运行。

6.2 堆栈工作原理
6.2.2 函数栈帧的构建

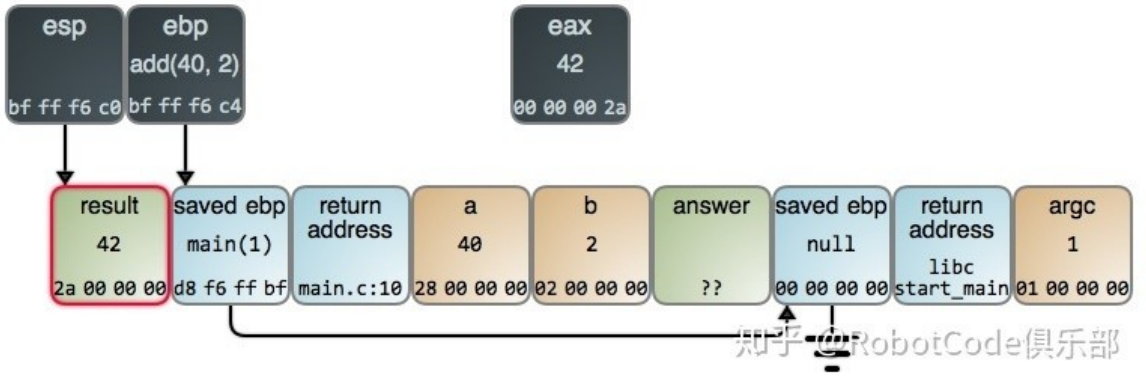
```
9.    subl $4, %esp # make room for result
```




```
      movl 12(%ebp), %eax # move b to eax
10.   movl 8(%ebp), %edx  # move a to edx
      addl %edx, %eax     # add edx into eax. total is 42.
```




```
11.   movl %eax, -4(%ebp) # copy eax to result
```

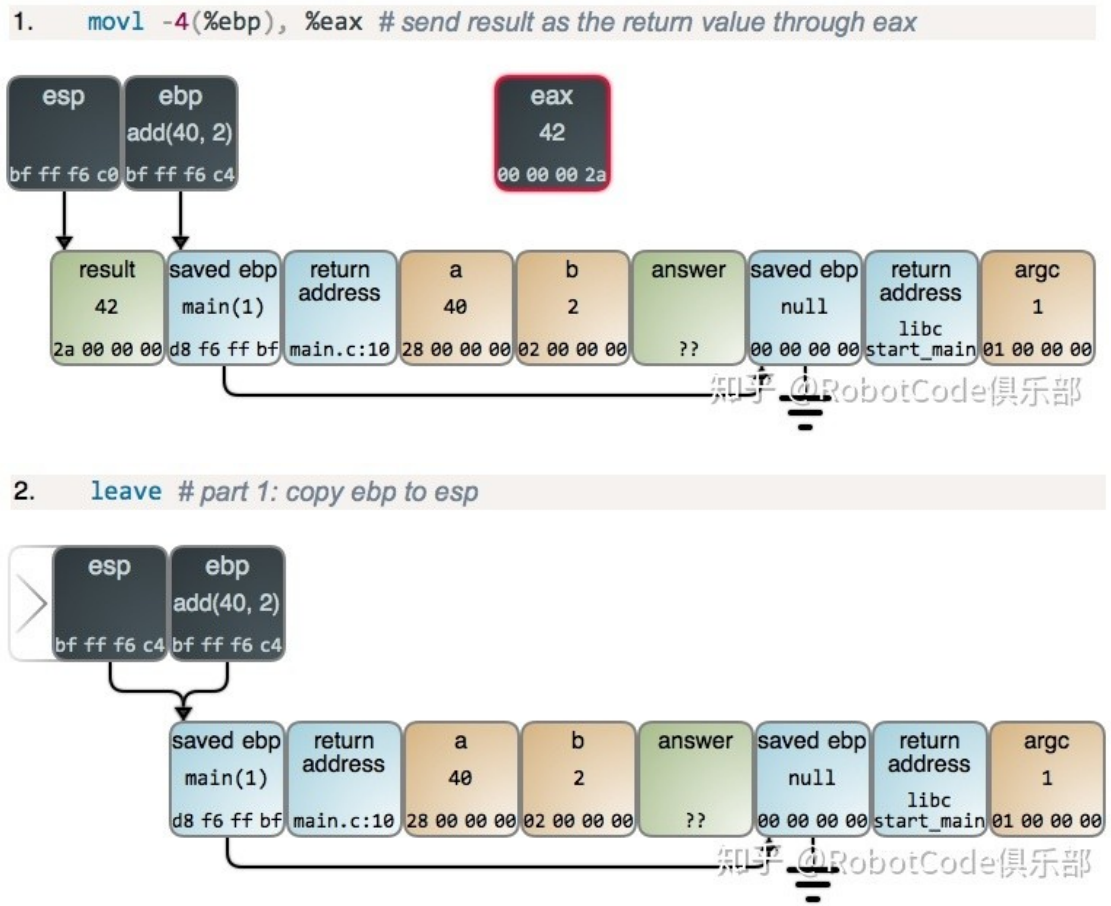


至此，add 完成了它的工作，add 函数执行完，堆栈操作将反向执行。

6.2 堆栈工作原理
6.2.3 函数栈帧的销毁

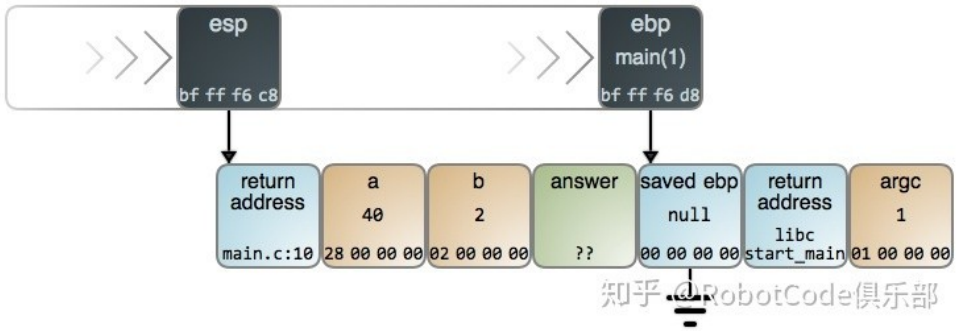
```
int add(int a, int b)
{
    int result = a + b;
    return result;
 }
int main(int argc)
 {
    int answer;
    answer = add(40, 2);
 }
```

在 a + b 赋值到 result 之后

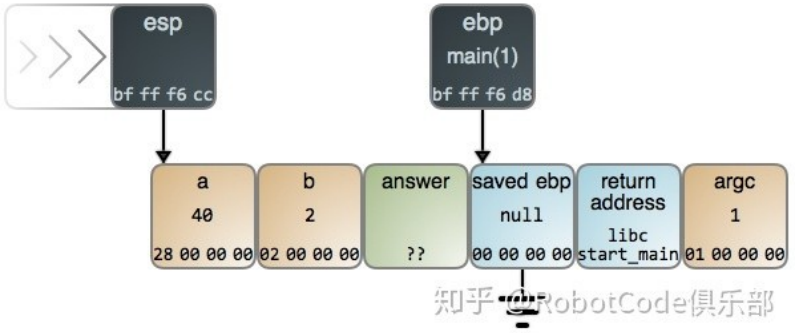


6.2 堆栈工作原理
6.2.3 函数栈帧的销毁

```
3.    leave  # part 2: pop into ebp
```



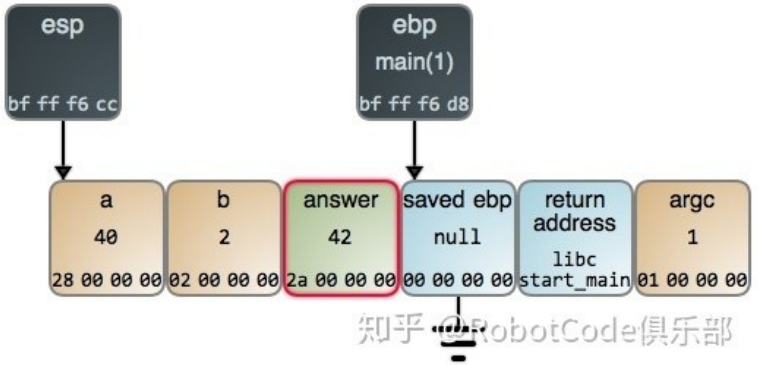

```
4.    ret  # pop into eip (instruction pointer)
```



优化关闭后， eax 等于 result 。然后 leave 指令运行，重置 esp 以指向当前栈帧的起点，然后恢复保存的 ebp 值。这两个操作在逻辑上是不同的，如图 2 ， 3 所示，但如果使用调试器进行跟踪，它们就会自动发生。

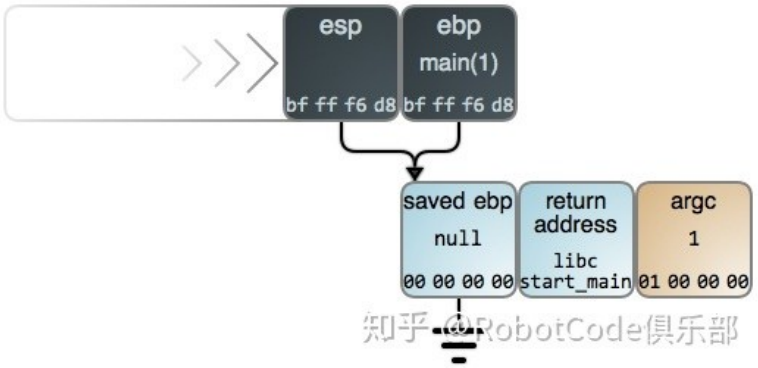
在 leave 运行之后，将恢复前一个栈帧。 **add 被调用的唯一痕迹，是堆栈顶部的返回地址。它包含 add 完成后必须运行的 main 指令的地址。** ret 指令负责处理它：它将返回地址弹出到 eip 寄存器中，该寄存器指向要执行的下一条指令。该程序现已返回 main 。

```
5.    movl  %eax, -4(%ebp)  # copy eax to answer
```

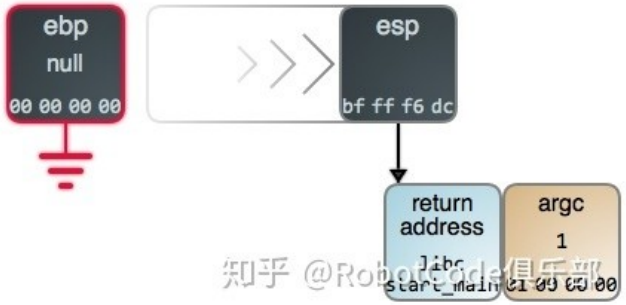


6.2 堆栈工作原理
6.2.3 函数栈帧的销毁

6. `leave` *# part 1: copy ebp to esp*



7. `leave` *# part 2: pop into ebp*



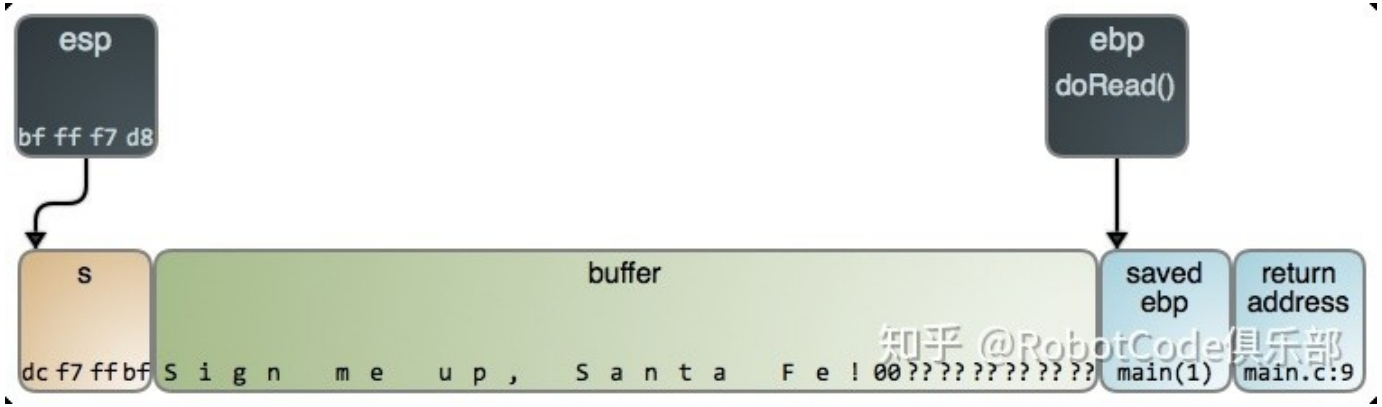
8. `ret` *# pop into eip (instruction pointer)*

main 将 add 的返回值复制到本地变量 answer 中，然后运行自己的“尾声”，这与任何其他栈桢被销毁的过程相同。 main 的唯一特性是保存的 ebp 为 null ，因为它是代码中的第一个栈帧。在最后一步中，执行流程转到 C 库 (libc) ，返回到操作系统。

6.3 堆栈缓冲区溢出

```
void doRead()
{
    char buffer[28];
    gets(buffer);
}
int main(int argc)
 {
    doRead();
 }
```

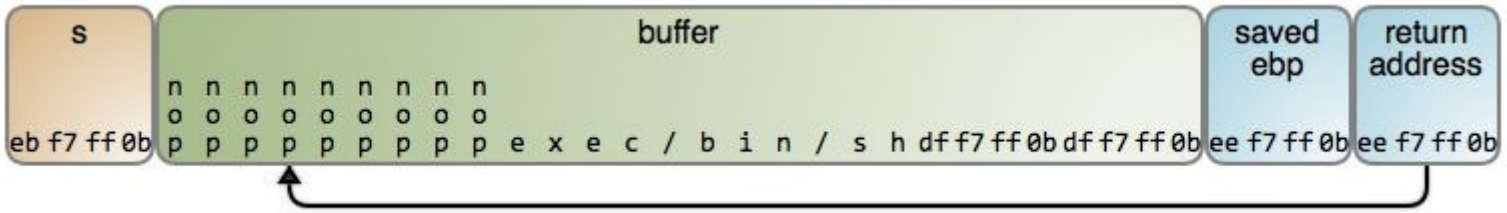
其从标准输入读取。 **gets** 函数一直读取，直到遇到换行符或文件结束。读取字符串后堆栈如图所示：



6.3 堆栈缓冲区溢出

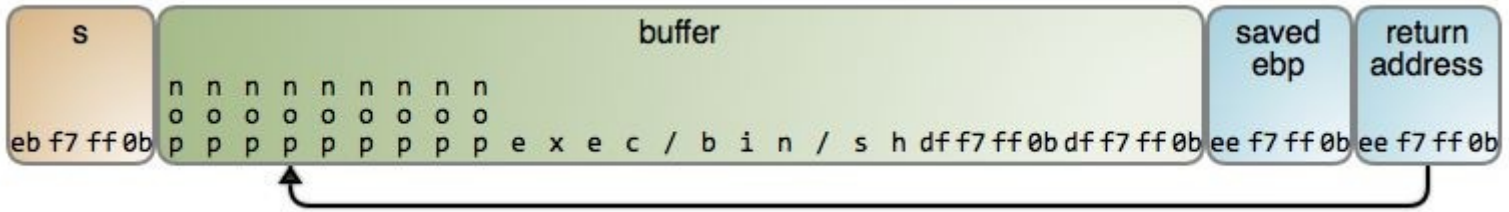
```
void doRead()
{
    char buffer[28];
    gets(buffer);
}
int main(int argc)
 {
    doRead();
 }
```

gets 不知道缓冲区的大小：它将继续读取输入并将数据填充到缓冲区之外的堆栈中，删除保存的 ebp 值、返回地址和下面的任何内容。为了利用这种漏洞，攻击者制造一个精确的“输入内容”并将其输入程序。堆栈在攻击后如图所示：



其基本思想是提供要执行的恶意程序代码，并覆盖堆栈上的返回地址以指向该代码。

6.3 堆栈缓冲区溢出



这个漏洞的有效“负载”有许多显著的特征。

1、它从几个 nop 指令开始，以增加恶意代码被成功执行的几率。这是因为返回地址是绝对的，必须猜测，因为攻击者不知道他们的代码将存储在堆栈的哪个位置。但是，只要它们停留在一个 nop 上，这个漏洞就会起作用：处理器将执行 nop ，直到它遇到可以工作的指令为止。

2、 exec /bin/sh 表示执行 shell 的原始汇编指令（例如，假设该漏洞位于一个联网程序中，因此该漏洞可能提供对系统的 shell 访问）。将汇编指令嵌入到一个程序中，使程序产生一个命令窗口或者用户输入。有时脆弱的程序会对其输入调用 tolower 或 toupper ，迫使攻击者编写不属于大写或小写 ascii 字母范围的汇编指令。因为如果改变大小，栈上的值就会改变，跟攻击者的预期不一致。

3、最后，攻击者将猜测的返回地址重复几次，以获得对他们有利的机会。通过从一个 4 字节的边界开始并提供多次重复，它们更有可能覆盖堆栈上的原始返回地址。

6.3 堆栈缓冲区溢出

值得庆幸的是，现代操作系统有许多针对缓冲区溢出的保护措施，包括非可执行堆栈和堆栈金丝雀。堆栈金丝雀如图所示：

金丝雀是由编译器实现。例如，GCC 的堆栈保护选项会导致在任何可能易受攻击的函数中使用金丝雀。函数序言将一个幻数加载到 canary 位置，而在函数调用后该值应该还是完整的。如果不是，则可能发生缓冲区溢出（或错误），并通过 __stack_chk_fail 中止程序。由于它们在堆栈上的战略位置，金丝雀使得利用堆栈缓冲区溢出变得更加困难。

谢谢大家！