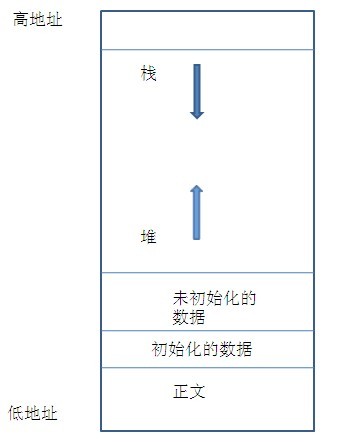
模块一：关于栈帧(stack frame)的学习

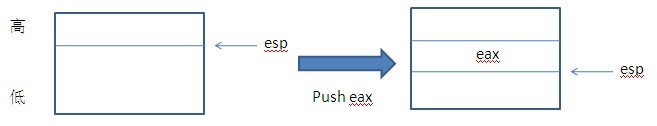
从逻辑上讲，栈帧就是一个函数执行的环境：函数参数、函数的局部变量、函数执行完后返回到哪里等等。栈是从高地址向低地址延伸的。每个函数的每次调用，都有它自己独立的一个栈帧，这个栈帧中维持着所需要的各种信息。寄存器ebp指向当前的栈帧的底部（高地址），寄存器esp指向当前的栈帧的顶部（低地址）。注意：EBP指向当前位于系统栈最上边一个栈帧的底部，而不是系统栈的底部。严格说来，“栈帧底部”和“栈底”是不同的概念;ESP所指的栈帧顶部和系统栈的顶部是同一个位置。

下面用一下CSDN上面yxysdcl老师的文章：

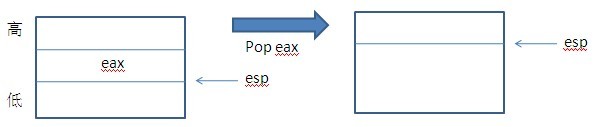


回顾一下入栈和出栈的操作：

入栈操作:push eax; 等价于 esp=esp-4,eax->[esp];



出栈操作:pop eax; 等价于 [esp]->eax,esp=esp+4;



通过一个函数我们来理解栈帧的变化：

void func(int m, int n) {

int a, b;

a = m;

b = n;

}

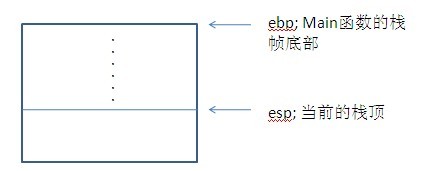
main() {

func(m, n);

L: 下一条语句

... }

在main调用func函数前，栈的情况，也就是说main的栈帧：

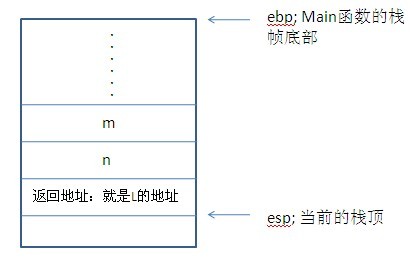


从低地址esp到高地址ebp的这块区域，就是当前main函数的栈帧.当main中调用func时，写成汇编大致是:

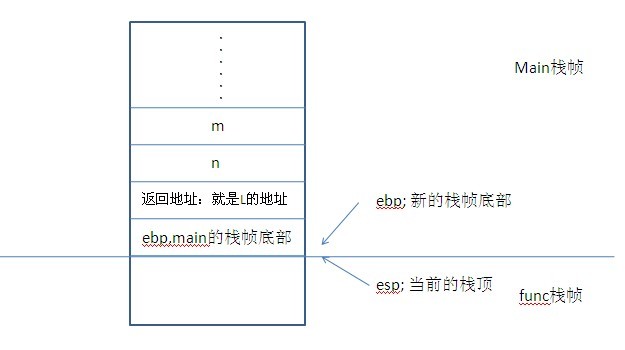
push m

push n; 两个参数压入栈

call func; 调用func，将返回地址填入栈，并跳转到func



跳转到func之后，我们就要 push ebp; 这个很重要，因为现在到了一个新的函数，也就是说要有自己的栈帧了，那么，必须把上面的函数main的栈帧底部保存起来，栈顶是不用保存的，因为上一个栈帧的顶部讲会是func的栈帧底部. （两栈帧相邻的）mov ebp, esp; 上一栈帧的顶部，就是这个栈帧的底部。

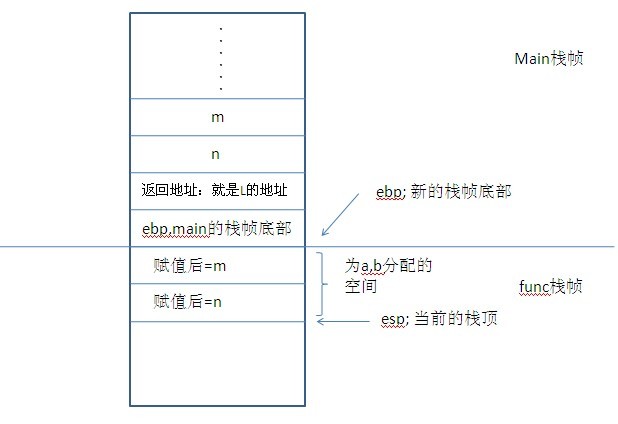


到了这一步，新的栈帧开始了。

sub esp, 8  int a, b 这里声明了两个int，所以esp减小8个字节来为a,b分配空间mov dword ptr [esp+4], [ebp+12]   a=m

mov dword ptr [esp], [ebp+8] b=n

这样以后栈就变成了这样：



ret 8 返回，然后8是什么意思呢，就是参数占用的字节数，当返回后，esp-8，释放参数m,n的空间

通过ebp，能够很容易定位到上面的参数。当从func函数返回时，首先esp移动到栈帧底部（即释放局部变量），然后把上一个函数的栈帧底部指针弹出到ebp,再弹出返回地址到cs:ip上，esp继续移动划过参数，这样，ebp,esp就回到了调用函数前的状态，即现在恢复了原来的main的栈帧。

然后栈帧表示程序的函数调用记录，而栈帧又是记录在栈上面，很明显栈上保持了n个栈帧的实体，就可以说栈帧将栈分割成了n个记录块，但是这些记录块大小不是固定的。因为栈帧不仅保存诸如:函数入参、出参、返回地址和上一个栈帧的栈底指针等信息，还保存了函数内部的自动变量（甚至可以是动态分配内存，alloca函数就可以实现，但在某些系统中不行），因此，不是所有的栈帧的大小都相同。

**其他的学习：**

（1）栈帧上的动态内存分配

    前面已经说明过一点：在大部分系统中，栈帧上可以进行动态内存的分配。malloc、calloc和realloc函数都是在堆上动态分配一块内存，在使用过后一定要记得释放动态分配的内存，否则就会产生内存泄露，最终降低系统的性能。

    但是如果要在栈帧上动态分配内存的话，那么在函数返回时会自动释放这些内存，而不必担心忘记释放动态分配的内存。我们知道在Linux内核中，每个进程的栈只有1-2个页的大小，即4K-8K大小，需要很珍惜的使用这部分空间；不过实用户栈的空间很大，可以随着需要动态的扩充，而不必担心栈不够用，因此我们还是可以放心的使用alloca动态分配函数在用户栈帧上分配内存。

（2）函数调用深度

    在很多系统中都对函数调用的深度做了限制，函数调用深度是指函数嵌套的程度。函数嵌套的程度决定了在栈上同一时刻所拥有的栈帧的最大数量，函数调用的嵌套程度对用户进程来说不是什么问题，但是在内核中栈的大小固定且不能重新分配，因此调用的深度在内核中就存在很大的意义，这里我们不做详述。

（3）函数调用的参数

    栈帧部分已经描述了函数参数的保存位置，即保存在调用者栈帧的尾部固定长度偏移位置，程序运行时就根据函数的定义和该位置取参数进行相应的运算。

注意：这里函数调用的参数显然存储在函数调用者的栈帧中，而不是被调用函数的栈帧中。

（4）栈的回溯

    学习编程和Linux内核的童鞋一定经常听到“栈的回溯”，它是指系统自主打印进程调用栈的行为。从上面描述栈帧的情况可以看出，系统在将栈打印出来的顺序应当是调用的反顺序，它是从esp（低地址）一点一点向高地址回溯，这正是栈帧形成的反过程。因此，我们经常从下到上看函数的调用，不过有些日志系统将导出的回溯信息重新排序，可以从上到下来查看函数调用顺序。

(5)基本介绍stack 栈

在32位系统中，堆栈每个数据单元的大小为4字节。小于等于4字节的数据，比如字节、字、双字和布尔型，在堆栈中都是占4个字节的；大于4字节的数据在堆栈中占4字节整数倍的空间。

(6)EBP的一些知识

EBP寄存器是用于访问堆栈中的数据的，它指向堆栈中间的某个位置，函数的参数地址比EBP的值高(估计是调用者的某个变量)，而函数的局部变量地址比EBP的值低，因此参数或局部变量总是通过EBP加减一定的偏移地址来访问的，比如，要访问函数的第一个参数为EBP+8。(估摸着调用者的某个变量)。

一般来说，参数都是从右往左入栈的。