在各种计算机编程语言中，都少不了函数调用来完成某些特定功能。我们要使用一个函数前，必定会先定义该函数（可能还会先声明它）。在这个过程中，我们大多会给这个函数传递某些‘参数’，当我们定义被调用函数时，函数名后的括号( )中即为我们传入的“形参”，函数可以在函数体中描述如何操作这些“形参”；同时，我们调用函数时，例如 fun(param1,param2,...),括号里面的这些param，我们称之为实参。简单来说，**调用函数时就是把实参传给定义的函数作为形参以供函数来使用它们**。

    形参和实参的定义确实很好理解，但如果仅仅是理解它们是什么的话，这往往是不够的。

    接下来，我会在系统堆栈层面说明一下函数调用过程中，形参与实参以及函数局部变量是如何在堆栈里变化的。

    先说一个C语言中的例子，如果我们想要通过一个子函数交换在主函数main()里面定义的两个局部变量x,y的值的话，一些初学者可能会像这样写code：



    写完之后，在主函数打印了一下x,y发现x与y的值根本没有变化啊！

    在这里，我们在主函数中定义了两个int类型的局部变量x和y，它们的值分别为3和5，然后我们调用了在主函数前面定义的交换函数Exchange，并将变量x与y分别作为了两个实参传递给它；再看Exchange函数中，我们的得到了两个传过来的形参a、b，然后我们通过同一个中间变量使它们（注意，两个形参）交换。得到的结果时，主函数的x，y的值并没有发生交换。

    这个代码正确的写法应该是：



    这次我们的实参是&x,&y，即传递过去的是x,y的地址,相当传递过程中存在int \*a=&x与int \*b=&y操作，其结果是让a指向x，b指向y。然后同样通过tmp完成值交换。

   如果我们单单从指针方面去考虑为什么这次能够交换成功，可能还不太容易弄懂交换的本质，接下来让我谈一谈这次交换在系统堆栈中是如何做到的。

    众所周知，计算机存储体系分为4层，它们的读写速度从慢到快依次是外存<内存<高速缓存<CPU寄存器。

    其中，寄存器又可分类为通用寄存器，段寄存器，辅助寄存器，两个特殊寄存器：IP寄存器和Flag标志位寄存器

    所有的C源程序都要进行编译，从而生成最终的机器指令代码和文件，那么，上述的内容最终是由编译软件负责完成的；也就是说，编译软件将我们的ASCII码形式的源程序代码，根据自己的原则，使用上述的寄存器完成源代码的任务。反过来说，如果在编程中需要特殊处理，那么，就需要给编译器一定的“指导”，方可达到我们的目的！

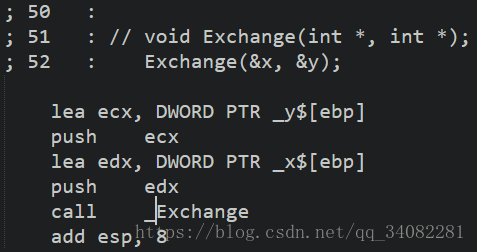
    辅助寄存器中有两个寄存器bp（基址寄存器）和sp （栈寄存器），通常bp表示系统堆栈的栈底地址，sp表示系统堆栈的栈顶地址。

    特殊寄存器IP（指令寄存器），可以用来存储“下一步”要执行的某个进程（线程）的代码的地址的；

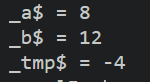
**函数调用的过程中就发生了如下的变化：**

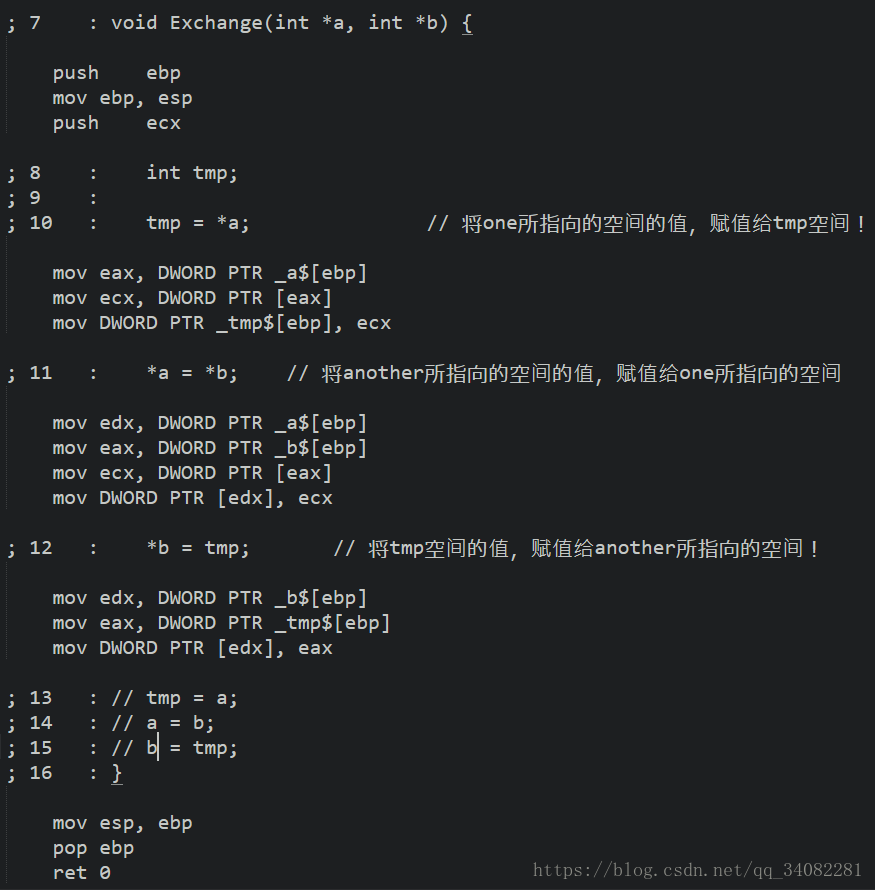
首先，我们将代码的执行过程显示成以.asm为后缀的汇编文件

https://img-blog.csdn.net/20180425124407628?watermark/2/text/aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3FxXzM0MDgyMjgx/font/5a6L5L2T/fontsize/400/fill/I0JBQkFCMA==/dissolve/70



可以看见，当我们调用Exchange函数时，我们将x与y的地址作为实参变量传递了过去，在汇编层面，发生的事情是先通过lea指令将ebp[x]和ebp[y]地址的偏移量，把它们分别传送到通用寄存器里的dx和cx寄存器里面。然后通过call指令调用函数Exchange.也就是说，调用函数时，我们将距离ebp（栈底地址）为4个和8个字节的x和y变量地址偏移量，传送给ecx和edx寄存器；有人可能想问，上图中为什么\_x$=-8,\_y$=-12。我们都知道，堆栈是一种后进先出的数据结构，局部变量定义后，会使堆栈内存减小，地址往低处走，局部变量相对于栈底指针bp在其上面，也就是堆栈的低处，所以偏移量用负值表示。同时，主函数其实也是一个**相对于操作系统的子函数，**操作系统调用主函数，然后主函数在调用子函数。也就是说，建立主函数时，我们在主函数中定义的相对于主函数的局部变量也同样被压入了保存变量空间的系统堆栈。接下来我将通过主函数调用子函数的例子说明：

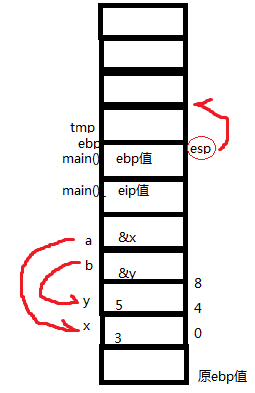




    子函数是被call指令执行的，call指令能够停止主函数的运行，并进入子函数，运行其函数体内容。当然，**还要保护主函数的运行状态信息，以使子函数调用完成返回后能够接着执行主函数后面的操作!**进入子函数后，先将ebp入栈，这样做是为了保护主函数的ebp信息。第二步则是将esp移到ebp，栈顶指针和栈底指针重合，形成了一个空栈。而这个空栈就是为Exchange函数中**局部变量**所准备的空间。Exchange函数不仅拥有传进来的形参变量a,b，还有自己定义的局部变量tmp。

第三步的push ecx，是为局部变量tmp准备的，这个操作后，esp向地址低处移动4B，相当于偏移量为-4；

与此同时，堆栈里的情况如下图所示：



    实参变量是调用函数Exchange时入栈的，而且是根据括号里的顺序($x,$y)从右往左依次入栈。而赋值给形参变量是从左往右赋值的，也就是上面所说的int \*a=&x与int \*b=&y。往上，我们还要push两个名为ebp和eip的值入栈，ebp入栈是因为我们要保存main原ebp的值，而为什么这里还有4B的eip值也入栈了呢？这是因为，eip值是用来存放“下一步"要执行的某个进程（线程）的代码的地址的，主函数调用子函数，主函数被”中断“，当子函数返回到主函数时，同时也应该准确地返回到主函数被中断的地方，继续往下执行。所以eip是保护”现场信息“的关键。于是，这也就解释了为什么形参变量a和b与ebp的偏移量分别为8B和12B了。如果子函数在其内部继续定义局部变量，那么空间就继续往上走，esp指针也跟着往上走就可以了。函数调用的入栈顺序就很明了了：实参表达式（形参）->main()eip->main()ebp->被调函数局部变量。如果子函数中再调用其它函数，就如printf(),入栈细节也是如法炮制的。

    但是，这好像依旧没搞清楚最开始的问题，即它们是如何成功交换值的。要搞清楚这个，我们就必须明白主函数局部变量，实参和形参三者的所占用空间关系。其重点就是，局部变量与实参并不是占用同一空间，然而**实参表达式算出来的值，在系统堆栈占用的空间，就是与之对应的形参变量的空间。**也就是说，如果我们按开头第一种代码写交换功能的话，我们交换的只是，形参变量空间中的值而已，因为它只相当于复制了一份主函数中x，y的值给形参而已，并没有产生能够改变真正在主函数中x,y的值。

但是如果我们传递给Exchagne的是x，y的地址的话，就相当于构建了Exchange中形参变量a和b与主函数x，y的一种地址指向”联系“。一旦我们建立了这种联系，我们就可以将a所指向空间的值（局部变量x的值）与b所指向空间的值(局部变量y的值)同过tmp进行交换。从而达成了交换的功能。

