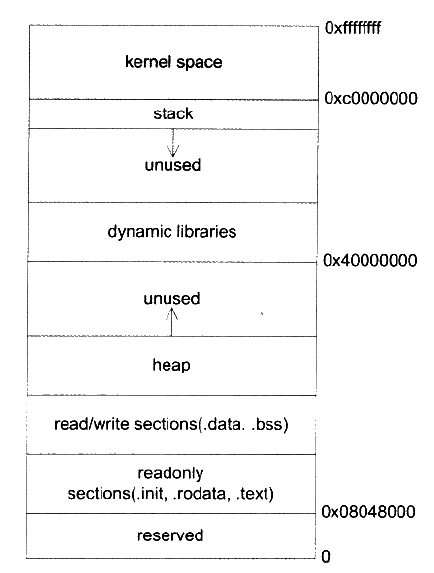
学习堆栈溢出，第一步肯定是要先了解栈的结构。栈用于维护函数调用的上下文，

没有栈就没有**函数**，以及**局部变量**......可见栈是不可或缺的，先上一张最经典的结构图



从上图可知栈是向下（**低地址方向**）增长的，所以栈底的初始地址是**0xbfffffff**，以后随着函数的调用，**ebp**会不断变化

假设程序是这样：

int main()

{

        int a;

        fun(a);

}

那么对应的流程如下，请结合下图理解^\_^

------------------------main-------------------------

[1]push a //参数a入栈

[2]call foo //push 当前地址，jmp foo

------------------------main-------------------------

-------------------------fun--------------------------

[3]push ebp //这里的ebp为main栈帧的栈底，用于待会返回前还原main的栈帧环境

[4]mov ebp,esp //既然保存完旧ebp了，那么ebp就立刻变身为foo函数的栈底，此时main的栈顶(esp)=foo的栈底(ebp)

[5]sub esp,0C0h //这里分配的空间用于保存ebx，esi，edi等寄存器值以及调试信息

[6]push ebx //保存ebx，esi，edi寄存器

push esi

push edi

[7]lea edi,[ebp-0C0h] //大家注意ebp-0C0h这片空间就是刚刚esp分配的空间，这段代码的作用是将

mov ecx,30h //这片空间全部赋值CC（调试信息），而这片空间对应的就是下图中的局部变量和其它数据，

mov ecx,0CCCCCCCCh //用户创建变量的同时将不断覆盖CC，如果用户一个指针变量的等于0xCCCCCCCC，

rep stos dword ptr es[edi] //那么编译器就认为指针没有经过初始化，给予警告，这里的0xCCCC就是大家最熟悉的'烫'，

//当然不同编译器不同规则，还有'屯'.......等等

[8]XXXXXXXXXXXXXXXXX //此处为函数的功能代码，在函数中，程序通过ebp-x的这样的方式得到局部变量的地址，

XXXXXXXXXXXXXXXXX //进而对变量实行操作

[9]pop edi //还原寄存器值

pop esi

pop ebx

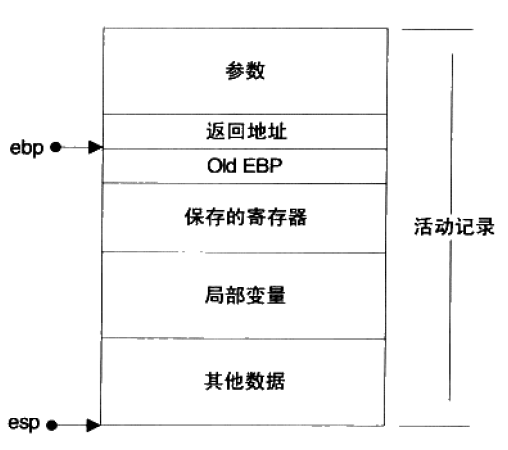
[10]mov esp,ebp //恢复进入函数前的esp和ebp

pop ebp

[12]ret //使用ret指令返回

-------------------------fun--------------------------

执行完上述代码后，栈就是酱紫的......



最后拓展一下.......函数的返回值，一般情况下，函数的返回值是存储在**eax**中进行传递的，那么大家就会有疑问，**eax**不够装怎么办？？

嗯嗯......这时候**eax**存储的是那个装不下的变量的**地址**，这地址有可能是堆上的（动态分配的变量）也有可能是栈上的（静态分配的变量）

OK，栈的主体流程就是以上这些，怎么样？并没有想象中的那么神秘吧^\_^

祝大家生活愉快~~