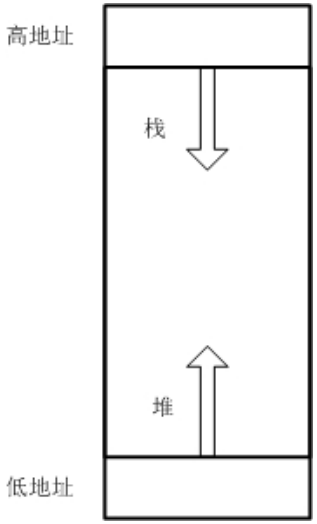
函数调用栈帧过程（带图详解）

2017年04月18日 15:32:35 cherishinging 阅读数 1827

函数调用我们大家都很熟悉，我们都知道函数调用是发生在栈上的。首先，我们来了解一下程序对内存使用的分区情况：

区域	作用
栈区（stack）	由编译器自动分配和释放，存放函数的参数值，局部变量的值等。操作方式类似与数据结构中的栈
堆区（heap）	一般由程序员分配和释放，若程序员不释放，程序结束时可能由操作系统回收。与数据结构中的堆是两码事，分配方式类似于链表
静态区（static）	全局变量和静态变量存放于此
文字常量区	常量字符串放在此，程序结束后由系统释放
程序代码区	存放函数体的二进制代码

那么什么是栈帧呢？百度百科的解释是：栈帧就是一个函数执行的环境。实际上，栈帧可以简单理解为：栈帧就是存储在用户栈上的（当然内核栈同样适用）每一次函数调用涉及的相关信息的记录单元。
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（地址地）。下图为典型的存取器安排，观察栈在其中的位置



在了解函数调用栈帧之前，我们先来认识几个寄存器：

寄存器名称	作用
eax	累加(Accumulator)寄存器，常用于函数返回值
ebx	基址(Base)寄存器，以它为基址访问内存
ecx	计数器(Counter)寄存器，常用作字符串和循环操作中的计数器
edx	数据(Data)寄存器，常用于乘除法和I/O指针
esi	源变址寄存器
dsi	目的变址寄存器
esp	堆栈(Stack)指针寄存器，指向堆栈顶部
ebp	基址指针寄存器，指向当前堆栈底部
eip	指令寄存器，指向下一条指令的地址

下面以一段程序为例介绍函数调用栈帧，

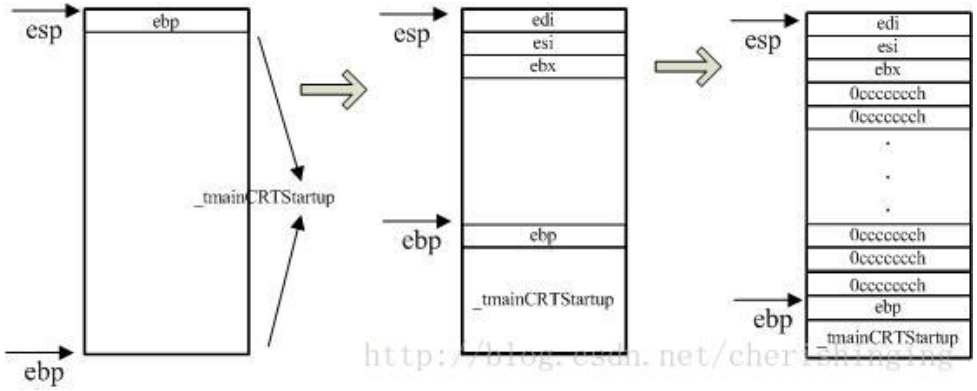
```c
#include <stdio.h>
int Add(int x,int y)
{
    int z = 0;
    z = x + y;
    return z;
}
int main()
{
    int a = 10;
    int b = 20;
    int ret = Add(a,b);

}
```

在VS2010下对程序进行调试，进入反汇编查看函数调用的具体过程，首先是函数的初始化，

```
000713F0  push        ebp
000713F1  mov          ebp,esp
000713F3  sub         esp,0E4h
000713F9  push        ebx
000713FA  push        esi
000713FB  push        edi
000713FC  lea         edi,[ebp-0E4h]
00071402  mov          ecx,39h
00071407  mov          eax,0CCCCCCCCh
0007140C  rep stos    dword ptr es:[edi]
```

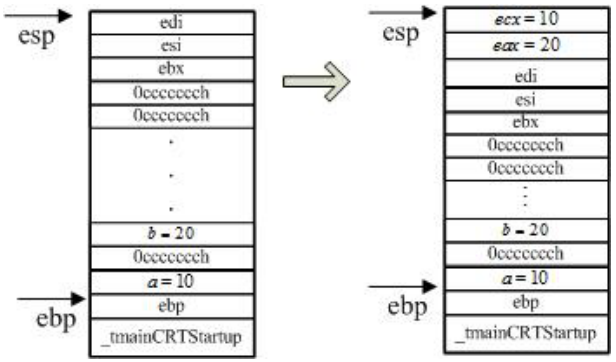
上述过程进行的操作由下图所示：



其实main（）函数是在_tmainCRTStartup函数中调用的，因此，首先创建main( )函数的栈帧，完成状态寄存器的保存，堆栈寄存器的保存，函数内存空间的初始化。

```
    int a = 10;
0007140E  mov         dword ptr [a],0Ah
    int b = 20;
00071415  mov         dword ptr [b],14h
    int ret = Add(a,b);
0007141C  mov         eax,dword ptr [b]
0007141F  push        eax
00071420  mov         ecx,dword ptr [a]
00071423  push        ecx
```

这里进行的操作是处理局部变量a, b，ret的创建，并将参数压栈，栈的变化如下图所示：

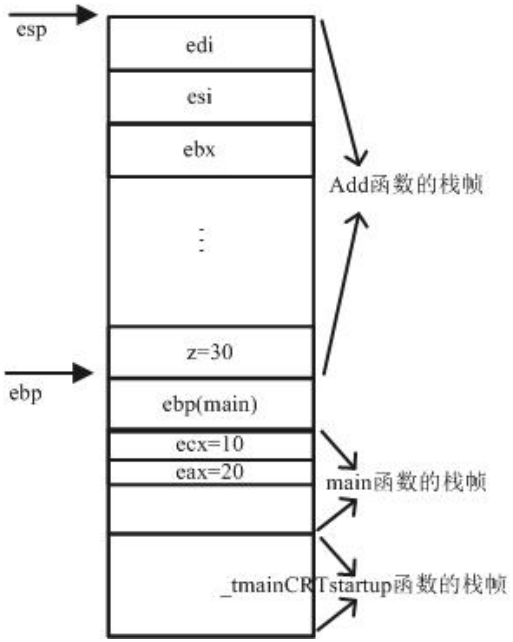


```
00071424  call        @ILT+215(_Add) (710DCh)
```

按下call指令的调用，先压栈call指令的下一条指令，然后跳转到Add函数的地方。按11进入Add函数的执行代码处。

```
000713A0  push        ebp
000713A1  mov         ebp,esp
000713A3  sub         esp,0CCh
000713A9  push        ebx
000713AA  push        esi
000713AB  push        edi
000713AC  lea         edi,[ebp-0CCh]
000713B2  mov         ecx,33h
000713B7  mov         eax,0CCCCCCCCh
000713BC  rep stos    dword ptr es:[edi]
    int z = 0;
000713BE  mov         dword ptr [z],0
    z = x + y;
000713C5  mov         eax,dword ptr [x]
000713C8  add         eax,dword ptr [y]
000713CB  mov         dword ptr [z],eax
    return z;
000713CE  mov         eax,dword ptr [z]
```

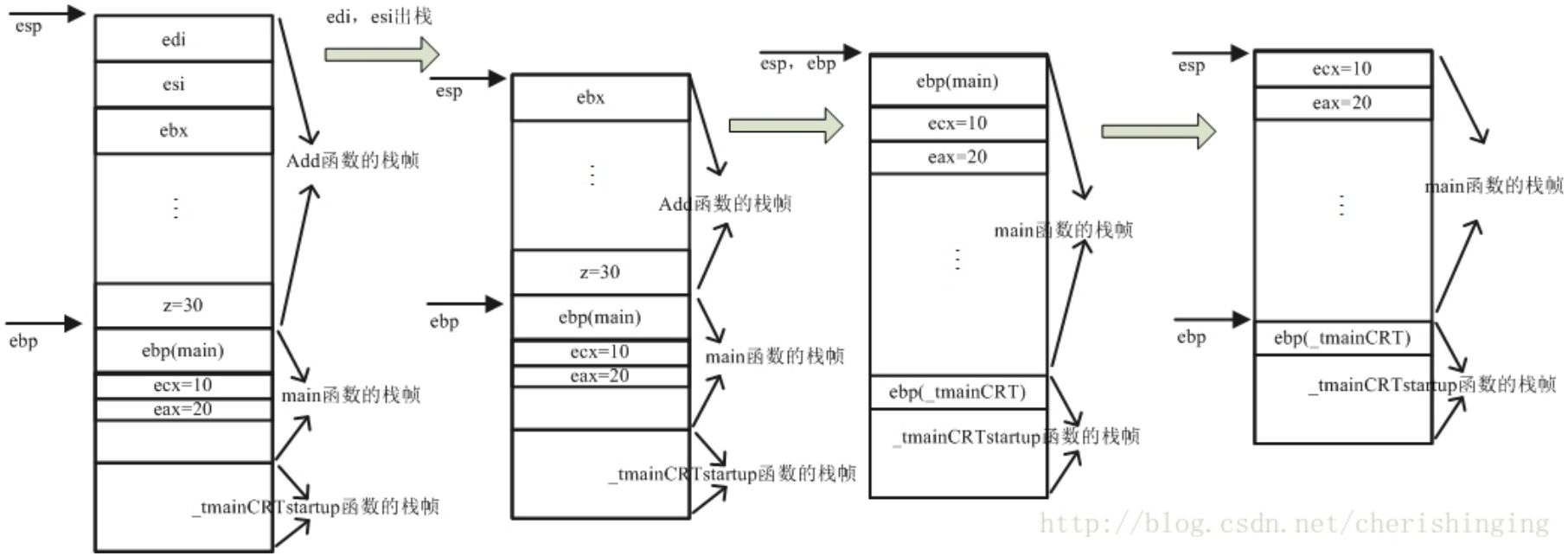
在000713c5处和000713c8分别在eax和ecx中取出形参a和b的值，并将他们相加。并把计算结果返回至eax中。
在此阶段的具体过程如下图所示：



接着，程序继续往下走：

```
1  000713D1  pop         edi
2  000713D2  pop         esi
3  000713D3  pop         ebx
4  000713D4  mov         esp,ebp
5  000713D6  pop         ebp
6  000713D7  ret
```

在这里首先edi, esi, ebx逐步出栈，每出栈一个，esp向下移动一步，接着将ebp的值赋给esp，也就是将esp移至ebp的位置，最后ebp出栈，将出栈的内容保存到ebp（此时ebp中保存的是main函数的ebp地址），回到main函数的栈帧。
注意：ret指令会使得出栈一次，并将出栈的内容当作地址，将程序执行跳转到该地址。



至此，Add函数调用完毕，完成了Add对Add函数栈帧的清栈。进入main函数的栈帧，采用类似的方式对main函数的栈帧进行清栈。回到_tmainCRTStartup函数的栈帧。

在了解了函数栈帧的调用过程后，我们来思考一下为什么要研究栈帧呢?
一个码农要是没遇见过coredump，那就幸运了。core file(coredump的转储文件)中保存的最重要内容之一，就是函数的call trace。还原这部分内容(栈回溯)，并与原代码对应上，尽快找出程序崩溃的位置和原因，是码农们一生的责任。当然，你如果有良好的开发环境和开发习惯，保留了现场环境（core file and lib file等）和unstrip的原程序，那么恭喜，也许你不用太费神，直接用GDB的backtrace功能，就可以找到症结所在。当然如果栈被冲掉了一部分，backtrace出来的就是一堆问号，要找出call trace就不容易了。这在缓冲区溢出时经常碰到。
总而言之，研究栈帧可以对内存管理有更深刻的认识。