琴鸟

c 函数调用产生的汇编指令和数据在内存情况(1)

一直对函数调用的具体汇编指令和各种变量在内存的具体分配，一知半解。各种资料都很详细，但是不实践，不亲自查看下内存总不能笃定。那就自己做下。

两个目的：

一，函数和函数调用编译后的汇编指令基本样貌

二，各种变量类型的内存状况。

一 函数和函数调用编译后的汇编指令基本样貌

1），空主函数

2），主函数调用,无返回直,无参函数.

3），主函数调用,无返回直,有参函数

昵称：琴鸟
园龄：7年8个月
粉丝：2
关注：1
+加关注

< 2016年11月 >

日	一	二	三	四	五	六
30	31	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	1	2	3
4	5	6	7	8	9	10

搜索

找找看

谷歌搜索

常用链接

3),主函数调用,有返回直,有参函数.

4) ,被调函数再调用函数.

二,各种变量类型的内存状况。

1).尝试 各种变量在全局或局部,或参数传递的情况.

2)常见语法的编译结果.

1),空主函数

代码:

```
int HariMain(void)
{
    return 0;
}
```

编译list:

```
 7                              [SECTION .text]
 8 00000000                             GLOBAL      _HariMain
 9 00000000                         _HariMain:
10 00000000 55                              PUSH      EBP
11 00000001 31 C0                           XOR       EAX,EAX
12 00000003 89 E5                           MOV       EBP,ESP
13 00000005 5D                              POP       EBP
```

我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

我的标签

c(1)
p民(1)
编译(1)
汇编(1)
内存(1)

随笔分类

.net(14)
c++(5)
好文转载
计算机系统
解惑(17)
烂尾的东西(2)
数据库(4)
算法(2)

随笔档案

2016年11月 (10)
2016年10月 (12)
2016年9月 (2)
2016年8月 (1)
2016年7月 (2)
2016年6月 (2)
2016年5月 (6)

14 00000006 C3 RET

Debug下观察寄存器和内存情况

无调用和数据。不需debug.

结论：c的空函数 最基本会有3条指令。

PUSH EBP

MOV EBP,ESP

POP EBP

2）,主函数调用,无返回直,无参函数

代码:

```
int HariMain(void)
{
    count();
    return 0;
}

void count()
{
    int a=1+2;
```


文章分类

最新评论

```
}
```

编译list:

```
     7                                  [SECTION .text]
     8 00000000                                 GLOBAL      _HariMain
     9 00000000                             _HariMain:
    10 00000000 55                                  PUSH      EBP
    11 00000001 89 E5                               MOV       EBP,ESP
    12 00000003 E8 00000004                           CALL      _count
    13 00000008 5D                                  POP       EBP
    14 00000009 31 C0                               XOR       EAX,EAX
    15 0000000B C3                                  RET
    16 0000000C                                 GLOBAL      _count
    17 0000000C                             _count:
    18 0000000C 55                                  PUSH      EBP
    19 0000000D 89 E5                               MOV       EBP,ESP
    20 0000000F 5D                                  POP       EBP
21 00000010 C3                                RET
```

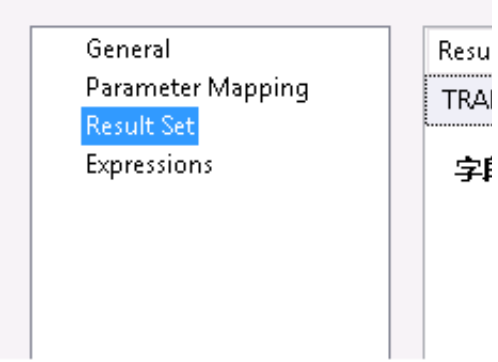
反汇编代码区(图1)

的存在形式。

基本数据类型.int ,char short.int a;a 标签代表一个地址的数据,里面的数据类型是int.所以占4个字节.a=3;给基本数据类型的标签赋直.就等于给标签代表的地址的数据赋直.a 标.....

--琴鸟

2. Re:sql for xml 嵌套



```
00280024: (                    ): push ebp                    ; 55
00280025: (                    ): mov ebp, esp                ; 89e5
00280027: (                    ): call .+4                    ; e804000000
0028002c: (                    ): pop ebp                     ; 5d
0028002d: (                    ): xor eax, eax                ; 31c0
0028002f: (                    ): ret                         ; c3
00280030: (                    ): push ebp                    ; 55
00280031: (                    ): mov ebp, esp                ; 89e5
00280033: (                    ): pop ebp                     ; 5d
00280034: (                    ): ret                         ; c3
```

代码执行前寄存器直

```
rax: 00000000_00000000 rcx: 00000000_00000000
rdx: 00000000_00000000 rbx: 00000000_00280000
rsp: 00000000_00310000 rbp: 00000000_00000000
rsi: 00000000_0008c2d0 rdi: 00000000_00300000
r8 : 00000000_00000000 r9 : 00000000_00000000
r10: 00000000_00000000 r11: 00000000_00000000
r12: 00000000_00000000 r13: 00000000_00000000
r14: 00000000_00000000 r15: 00000000_00000000
rip: 00000000_00000024
eflags 0x00000042: id vip vif ac vm rf nt IOPL=0 of df if tf sf ZF af PF CF
```

代码执行前堆栈情况.

```
Stack address size 4
| STACK 0x00310000 [0x00000000]
| STACK 0x00310004 [0x00000000]
| STACK 0x00310008 [0x00000000]
| STACK 0x0031000c [0x00000000]
| STACK 0x00310010 [0x00000000]
| STACK 0x00310014 [0x00000000]
```

根据汇编指令大概理解和实验观察点:

1) 主函数执行call之后查看栈

--琴鸟

3. Re:转 快速建立Subversion

svn,地址

svn://ip/svn

--琴鸟

4. Re:转 快速建立Subversion

和vs 配合 如何 给所有文件加上 lock 属性。先直接加入 项目到 svn。这个时候没有加锁转到 项目根目录。右键 , svn 菜单。属性。新加入 needs-lock 属性。关闭 vs。开启 vs.....

--琴鸟

5. Re:转 快速建立Subversion

1. 安装 SubversionC:\Program Files\Subversion2.建立目录存放文档数据E:\project\svnproject是我们所有项目的文档目录。svn是我们第一个项.....

--琴鸟

阅读排行榜

```
Stack address size 4
  | STACK 0x0030fff8 [0x0000002c]
  | STACK 0x0030fffc [0x00000000]
  | STACK 0x00310000 [0x00000000]
  | STACK 0x00310004 [0x00000000]
```

发现 已经有2条4字节数据。根据 图1代码，

Push ebp ：压入 ebp 0x00000000

Call .+4 效果如：push eip jmp near ptr 标号

所以栈顶的数据就是 被调函数返回时的指令地址 0x0000002c

同时ip ，指令地址变为 被调函数地址。

Call 指令 是用偏移 数字来表示 指令地址 。这里是.+4.

```
(0) [0x000000280027] 0010:0000000000000027 (unk. ctxt): call .+4 (0x00280030)
  ; e8040000000
<bochs:16> s
Next at t-26310387
rax: 00000000_00000000 rcx: 00000000_00000000
rdx: 00000000_00000000 rbx: 00000000_00280000
rsp: 00000000_0030fff8 rbp: 00000000_0030fffc
rsi: 00000000_0008c2d0 rdi: 00000000_003d0000
r8 : 00000000_00000000 r9 : 00000000_00000000
r10: 00000000_00000000 r11: 00000000_00000000
r12: 00000000_00000000 r13: 00000000_00000000
r14: 00000000_00000000 r15: 00000000_00000000
```

2）被调函数执行ret 之后 查看栈

执行ret指令,cpu会自动执行效果同样的1条指令。

Pop eip <eip=adr(ss,esp);esp=esp+4>

1. 关于URL编码/javascript/js url 编码(轉)(3984)
2. 五彩珠游戏(2446)
3. repeater 的编辑功能(1778)
4. 字符编码(1689)
5. Format函数(转)(1400)

评论排行榜

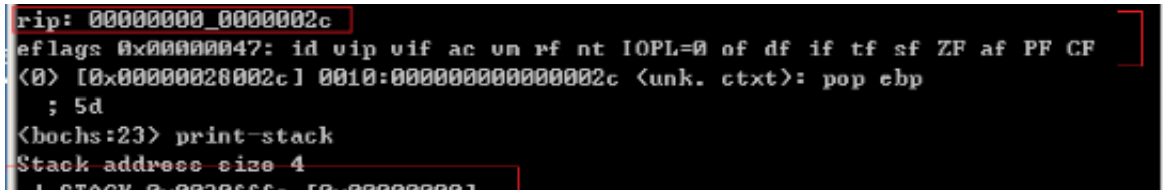
1. 五彩珠游戏(7)
2. .net后台通过xmlhttp 和远程服务通讯(5)
3. XMLHttpRequest介绍(5)
4. 转 快速建立Subversion(4)
5. 自定义控件(输入框,数字)(4)

推荐排行榜

1. p民和猫(3)
2. 五彩珠游戏(2)
3. 角色权限模块(1)

也就是指令地址重回 调用函数call 之后的下一指令的地址。同时栈顶地址向高地址移动。

确实如此。Ip 已经是0x0000002c ，也就是call 之后的下一个指令地址。



结论：无参，无返回直。

依靠 成对的call ret 指令,来调用函数和返回。

Call： push eip ，

jmp near ptr 标号

把call 之后的指令压栈。再jmp 到 被调函数的内存地址。

Ret： pop eip

返回调用者。

3),主函数调用 无返回直,有参数函数

代码:

```
void count(int a,int b);
int HariMain(void)
{
```

```
    count(1,2);
    return 0;
}

void count(int a,int b)
{
    int c=a+b;
}
```

编译list:

```
7                               [SECTION .text]
 8 00000000                             GLOBAL      _HariMain
 9 00000000                         _HariMain:
10 00000000 55                              PUSH      EBP
11 00000001 89 E5                           MOV       EBP,ESP
12 00000003 6A 02                           PUSH      2
13 00000005 6A 01                           PUSH      1
14 00000007 E8 00000004                        CALL      _count
15 0000000C 31 C0                           XOR       EAX,EAX
16 0000000E C9                              LEAVE
17 0000000F C3                              RET
18 00000010                                 GLOBAL      _count
19 00000010                             _count:
20 00000010 55                              PUSH      EBP
```

21 00000011 89 E5 MOV EBP,ESP

22 00000013 5D POP EBP

23 00000014 C3 RET RET

反编译代码区内存数据：

```
00280024: (                    ): push ebp                      ; 55
00280025: (                    ): mov ebp, esp                  ; 89e5
00280027: (                    ): push 0x00000002               ; 6a02
00280029: (                    ): push 0x00000001               ; 6a01
0028002b: (                    ): call .+4                      ; e8040000000
00280030: (                    ): xor eax, eax                  ; 31c0
00280032: (                    ): leave                         ; c9
00280033: (                    ): ret                           ; c3
00280034: (                    ): push ebp                      ; 55
00280035: (                    ): mov ebp, esp                  ; 89e5
00280037: (                    ): pop ebp                       ; 5d
```

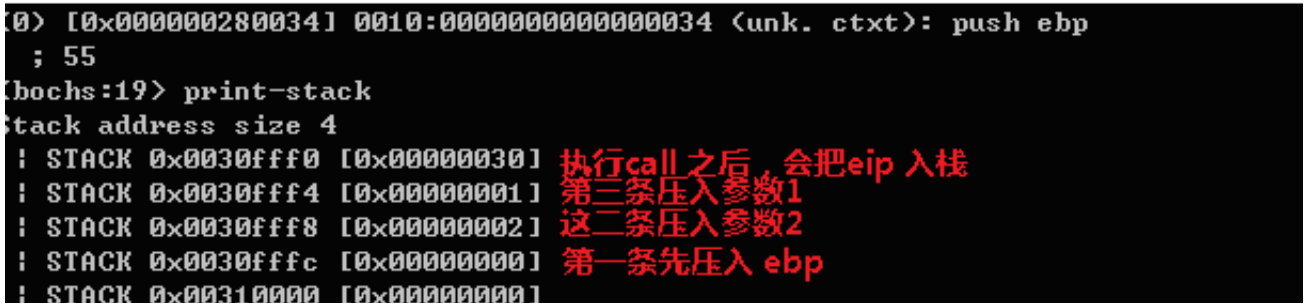
根据汇编指令大概理解和预测实验点:

1)带参,调用者会把参数入栈. 查看栈数据

调用前 参数确实入栈

```
(0) [0x00000028002b] 0010:000000000000002b (unk. ctxt): call .+4 (0x00280034)
  ; e8040000000
<bochs:15> trace-reg on
Register-Tracing enabled for CPU0
<bochs:16> print-stack
Stack address size 4
 | STACK 0x0030fff4 [0x00000001]  第三条压入参数1
 | STACK 0x0030fff8 [0x00000002]  这二条压入参数2
 | STACK 0x0030fffc [0x00000000]  第一条先压入 ebp
```

执行call，堆栈如之前实验，继续push eip



2)被调者如何使用参数

因为函数无返回直，并且函数计算的结果，并没有在任何地方使用。编译器直接机智的忽视掉被调函数的所有代码的编译。

结论：确实如 汇编语言 书上所讲。参数入栈是最后的参数先入栈。

4），主函数调用,有返回直,有参函数.

代码:

```
int count(int a,int b);

int HariMain(void)
```

```
{
    volatile int sum =count(1,2);
    return 0;
}

int count(int a,int b)
{
    int c=a+b;
    return c;
}
```

编译list:

```
 7                                  [SECTION .text]
 8 00000000                                GLOBAL     _HariMain
 9 00000000                            _HariMain:
10 00000000 55                               PUSH     EBP
11 00000001 89 E5                            MOV      EBP,ESP
12 00000003 50                               PUSH     EAX
13 00000004 6A 02                            PUSH     2
14 00000006 6A 01                            PUSH     1
15 00000008 E8 00000007                         CALL     _count
16 0000000D 89 45 FC                         MOV      DWORD [-4+EBP],EAX
```

```
    17 00000010 31 C0                    XOR     EAX,EAX
    18 00000012 C9                       LEAVE
    19 00000013 C3                       RET
    20 00000014                          GLOBAL  _count
    21 00000014                      _count:
    22 00000014 55                       PUSH    EBP
    23 00000015 89 E5                    MOV     EBP,ESP
    24 00000017 8B 45 0C                 MOV     EAX,DWORD [12+EBP]
    25 0000001A 03 45 08                 ADD     EAX,DWORD [8+EBP]
    26 0000001D 5D                       POP     EBP
27 0000001E C3                       RET
```

```
00280024: (                    ): push ebp                      ; 55
00280025: (                    ): mov ebp, esp                  ; 89e5
00280027: (                    ): push eax                      ; 50
00280028: (                    ): push 0x00000002               ; 6a02
0028002a: (                    ): push 0x00000001               ; 6a01
0028002c: (                    ): call .+7                      ; e807000000
00280031: (                    ): mov dword ptr ss:[ebp-4], eax ; 8945fc
00280034: (                    ): xor eax, eax                  ; 31c0
00280036: (                    ): leave                         ; c9
00280037: (                    ): ret                           ; c3
00280038: (                    ): push ebp                      ; 55
00280039: (                    ): mov ebp, esp                  ; 89e5
0028003b: (                    ): mov eax, dword ptr ss:[ebp+12] ; 8b450c
0028003e: (                    ): add eax, dword ptr ss:[ebp+8] ; 034508
00280041: (                    ): pop ebp                       ; 5d
```

1）上次实验只验证了参数入栈的情况。这次要看被调者如何使用参数。

使用 MOV EAX,DWORD [12+EBP]

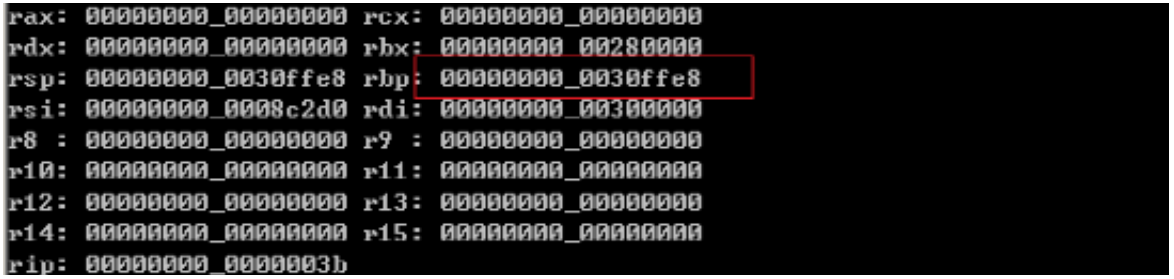
Add EAX,DWORD [8+EBP]

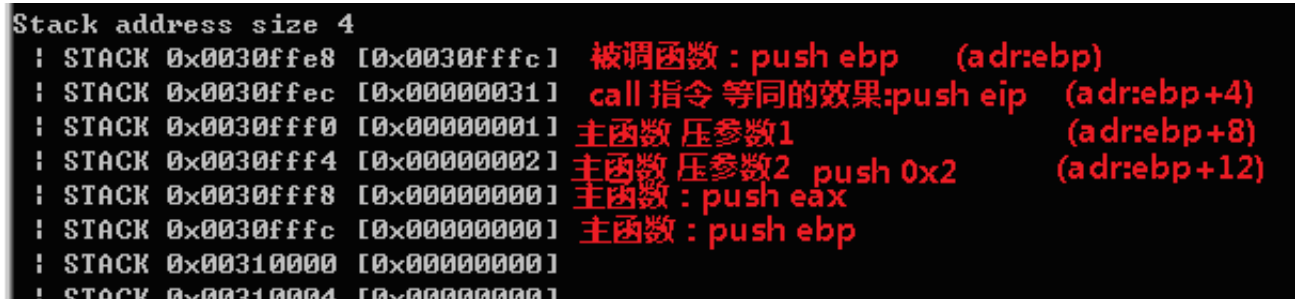
来取得实参。

为什么是这样。直接看图1。因为mov ebp esp.

Ebp 的直为0x30ffe8. 也就是指向当时的栈顶。

再看图二DWORD [12+EBP] 就是参数2。



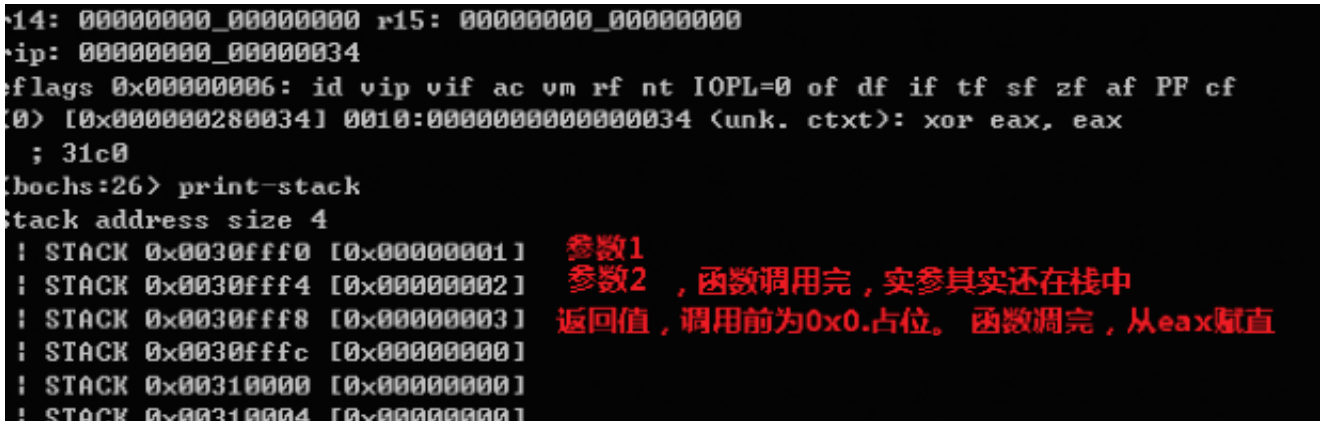


2）被调者如何取得返回值

MOV DWORD [-4+EBP],EAX

从寄存器eax 获得返回值，并给预先空处的栈的某个位置赋直（实参的后面地址）

执行 MOV DWORD [-4+EBP],EAX .后的寄存器和堆栈 情况。



结论：

C 编译器，被调函数一般会把返回值先放到寄存器eax 中。

调用函数需要返回直时，又会从eax 放入栈中，给栈中的某个地址赋直的形式，实参的后面（预先留了位置）

疑问：

为什么函数调用完， sp的值没有被修改？栈没有清空，还保留实参？

原来直观的觉得函数调用完，栈应该有 指令去退栈。

恩，恩，

函数运行时，入栈和出栈是代码本身实现的。

要保留一个值就push。

要使用或恢复就pop。

使用的过程就已经出栈了啊。

额，额。

那有没有一些数据是函数本身的数据呢？比如 常量，这个总要储存把。用完就要丢掉把。

恩，先测试函数的局部数据，看看编译后在内存中是怎么回事，会不会清栈。

5）被调函数有局部变量

代码:

```
int count(int a,int b);
int HariMain(void)
{
    volatile int sum =count(1,2);
    io_hlt();
}
```

```
int count(int a,int b)
{
    int c;
    int arrayint[2]={5,10};
    int i;
    for(i=0;i<2;i++)
    {
        c=c+arrayint[i];
    }
    c=c+a+b;
    return c;
}
```

编译list:

```
 8                                  [SECTION .text]
 9 00000000                                GLOBAL     _HariMain
10 00000000                            _HariMain:
11 00000000 55                              PUSH     EBP
12 00000001 89 E5                           MOV      EBP,ESP
13 00000003 50                              PUSH     EAX
14 00000004 6A 02                           PUSH     2
15 00000006 6A 01                           PUSH     1
```

```
16 00000008 E8 0000000A                  CALL    _count
17 0000000D 89 45 FC                     MOV     DWORD [-4+EBP],EAX
18 00000010 E8 [00000000]                CALL    _io_hlt
19 00000015 C9                           LEAVE
20 00000016 C3                           RET
21 00000017                              GLOBAL  _count
22 00000017                          _count:
23 00000017 55                           PUSH    EBP
24 00000018 89 E5                        MOV     EBP,ESP
25 0000001A 52                           PUSH    EDX
26 0000001B 52                           PUSH    EDX
27 0000001C 8D 4D FC                     LEA     ECX,DWORD [-4+EBP]
28 0000001F 8D 55 F8                     LEA     EDX,DWORD [-8+EBP]
29 00000022 C7 45 F8 00000005            MOV     DWORD [-8+EBP],5
30 00000029 C7 45 FC 0000000A            MOV     DWORD [-4+EBP],10
31 00000030                          L7:
32 00000030 03 02                        ADD     EAX,DWORD [EDX]
33 00000032 83 C2 04                     ADD     EDX,4
34 00000035 39 CA                        CMP     EDX,ECX
35 00000037 7E F7                        JLE     L7
36 00000039 03 45 08                     ADD     EAX,DWORD [8+EBP]
37 0000003C 03 45 0C                     ADD     EAX,DWORD [12+EBP]
38 0000003F C9                           LEAVE
```

39 00000040 C3 RET

根据汇编指令大概理解和预测实验点:

1) 这次被调函数，有一个局部变量。Int 的数组。

直接查看 被调函数返回前的堆栈情况把。

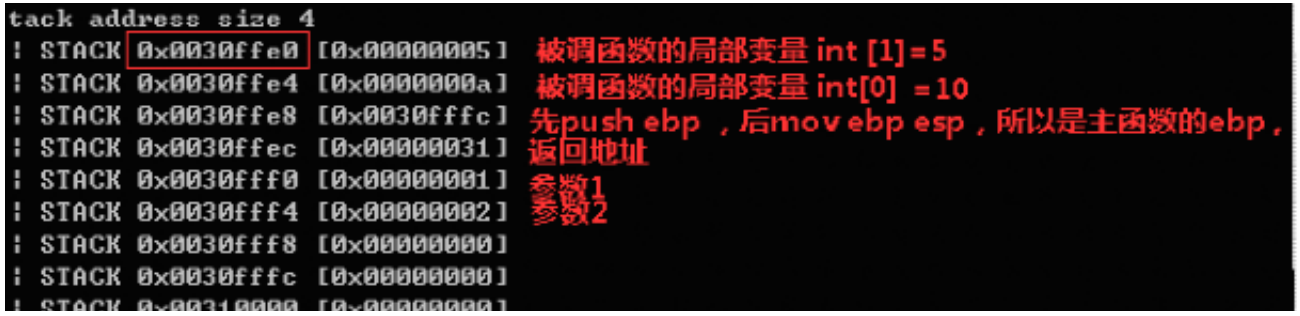
PUSH EDX

PUSH EDX

(上2条只想达到效果 sub esp 8？？)

MOV DWORD [-8+EBP],5

MOV DWORD [-4+EBP],10



2) 被调函数执行指令LEAVE后

Leave 等同

MOV SP,BP

POP BP

没有了被调函数的局部变量。

```
Stack address size 4
 | STACK 0x0030ffec [0x00000031] 返回地址
 | STACK 0x0030fff0 [0x00000001] 参数1
 | STACK 0x0030fff4 [0x00000002] 参数2
 | STACK 0x0030fff8 [0x00000000]
 | STACK 0x0030fffc [0x00000000]
```

只是修改了esp。也就是只移动了栈顶位置。

3）Ret 后

```
rax: 00000000_00000012 rcx: 00000000_0030ffe4
rdx: 00000000_0030ffe8 rbx: 00000000_00280000
rsp: 00000000_0030fff0 rbp: 00000000_0030fffc
rsi: 00000000_0008c2d0 rdi: 00000000_00300000
r8 : 00000000_00000000 r9 : 00000000_00000000
r10: 00000000_00000000 r11: 00000000_00000000
r12: 00000000_00000000 r13: 00000000_00000000
r14: 00000000_00000000 r15: 00000000_00000000
rip: 00000000_00000031
```

```
Stack address size 4
 | STACK 0x0030fff0 [0x00000001] 参数1
 | STACK 0x0030fff4 [0x00000002] 参数2
 | STACK 0x0030fff8 [0x00000000]
 | STACK 0x0030fffc [0x00000000]
 | STACK 0x00310000 [0x00000000]
 | STACK 0x00310004 [0x00000000]
```

LEAVE：释放当前子程序在堆栈中的局部变量,使BP和SP恢复成最近一次的ENTER指令被执行前的值。

MOV SP,BP

POP BP

哦。看到leave。完美解释了上一个疑问。

有局部变量的函数。用leave 指令。会修改 sp 。mov sp ，bp。

（后面测试，也可以不用leave，直接ADD ESP,16，简单达到修改栈顶地址目的）

也就是修改栈顶位置来达到 调用完函数后，栈丢弃被调者的局部变量。

结论：c 编译器，

1）调用函数时，函数有局部变量，会通过

sub esp xxx.

MOV DWORD [-8+EBP],yyy

达到把函数局部数据压栈的效果

2）当返回时，用leave 或ADD ESP,16 改变栈顶地址来达到清栈的效果。

关于c函数每次都有3条这样的指令

PUSH EBP

MOV EBP,ESP

。。。。。。。

POP EBP

的理解。

可能不太正确。

函数的调用，

首先代码方面

代码的进入函数和退出函数，因为有call 和ret 成对出现。所以没有问题。

那么数据方面呢。

假如A是一个函数，又假如我们使用esp来定位所有非静态数据。A(int x,int y)

用esp 栈顶代表返回地址。

用 esp+4代表第一个参数x

只要碰到变量x。编译器就用esp+4来替换。

但是假如a有局部变量。栈顶随便在变。Esp+4就不再是实参x了。

所以我们假如一进入函数就把esp 赋直给一个寄存器呢，比如和ebp。

那么ebp+4，就永远代表参数x了。我们可以把函数一些固定的数据放入栈中（参数，返回地址，函数的返回直）。用ebp+x 的形式来表示这些直，一些程序运行中可以改变大小的变量，如char * c;那就保留一个它的地址。而实际数据放入堆中。

而用ebp-x 来代表局部数据（所以编译器会把局部数据的代码往前编译出来？不管你定义局部变量的代码写在那里，总是放到临时变量push栈之前。以防之后有push临时变量的 指令？）。

那么函数a就可以准确找到包括参数和局部变量的所有数据。一个寄存器就解决了所有问题？

我想把ebp 叫做“准绳线”。

如果A 调用c函数,c函数和a一样聪明。一进入函数

1，push ebp ,先把a的ebp“绳子”放入栈中。

2，mov ebp,sp 把esp 赋直给ebp寄存器，建立c自己的"绳子"

那么c也可以和a 一样准确找到包括参数和局部变量的所有数据.

1)[ebp] 储存 a 的ebp。

2)[epb+4] 储存 返回地址。

3) [ebp+8] 储存 第一个参数

4)[epb-4] 储存 第一个局部数据。

当c返回时，清栈，从那里开始清呢。慢着。Ebp那么重要。首先要把a 的ebp"绳子" 找回来啊。

这个时候ebp 可是c的"绳子"(mov ebp ,esp)。那么a的ebp呢，不能丢啊，那么重要，记得有(push ebp　mov ebp ,esp 两条死都不分开好基友指令). 所以如果我们mov esp ，ebp,那么就达到了清栈的效果，而且现在栈顶的数据就是a的ebp"绳子"了。 那么我们再继续POP eBP,呵呵，寄存器ebp就是a 最重要的ebp了。执行ret吧，现在栈顶是a的返回地址了。我们回到了a 。而且ebp，寄存器保留了对a来说最重要的 准绳地址。

MOV SP,BP　POP BP 也是另外2条不分开的好基友指令，所以刚催有合体技能，leave指令。

慢慢记得好像,编译原理是有一个活动记录,这个ebp就是里面提到的top_sp

所以大概流程：

参数入栈，返回地址入栈， ebp入栈放入调用者的绳子，mov ebp,sp设置被调者的绳子

有局部变量，改变sp, sum sp sizeofvar.

用mov [ebp-x],aaa .来压入数据到栈。

最后，有函数局部数据采用 leave (mov sp,ebp 用被调者的绳子来清栈 ；pop ebp找回准绳) 或者
ADD　　ESP,16。。。。

结束 ret 。清掉当时压入的返回地址，最后搞了一圈回来只保留了实参，返回到调用函数代码。

函数一个一个嵌套调用的话，栈的数据越来越多。但当一个函数返回到上一个函数时，栈顶只保留被调函数的参数而已。一层一层返回，栈最终只有主函数的数据和主函数本身直接调用的函数的参数。

但是想到这里，因为每次清栈，都会保留调用函数的参数。那么a调b。回来，a再调c。那不是栈会保留b和c的参数？

Ok，验证一下。

HariMain 调用count和fint1

```
int count(int a,int b);
int fint1 (int a,int b);
int HariMain(void)
{
    volatile int sum =count(1,2);
    sum=sum+fint1(5,6);
    io_hlt();
}

int count(int a,int b)
{
    int c;
    int arrayint[3]={5,10,12};
    int i;
```

```
    for(i=0;i<3;i++)
    {
        c=c+arrayint[i];
    }
    c=c+a+b;
    int int1=fint1(1,2);
    c=c+int1;
    return c;
}

int fint1 (int a,int b)
{
    return a+b;
}
```

果然，回到a的时候。栈比调用前多了4个int 数据。

```
Stack address size 4
 ¦ STACK 0x0030ffe8 [0x00000005]
 ¦ STACK 0x0030ffec [0x00000006]
 ¦ STACK 0x0030fff0 [0x00000001]
 ¦ STACK 0x0030fff4 [0x00000002]
 ¦ STACK 0x0030fff8 [0x00280021]
 ¦ STACK 0x0030fffc [0x00000000]
 ¦ STACK 0x00310000 [0x00000000]
 ¦ STACK 0x00310004 [0x00000000]
```

没想到，之前老想不太明白的东西，边写边做，边反问。自己清晰了不少。

所以有最后一个疑问，c 编译器为什么栈要保留实参？

不可以先把返回地址入栈，再入参数？

那么指令可以用

mov sp ebp,pop ebp

add sp (由编译器计算所有参数的size)

ret .

这样不是更干净吗？

ebp 寄存器的值是当前函数的栈基址.而栈基址的里面的数据是调用者的栈基址.

就是说在内存的一个地址,也就是当前函数的栈基址写入了调用者的栈基址.

ebp寄存器的数据，一直都是当前函数的栈基址。一般（$ebp）+8 <寄存器的数据+8> 就是第一个参数。（$ebp）-偏移地址就是局部变量。

而 *（$ebp）〈寄存器的数据再到内存求数据〉 是 调用者的栈基址，所以被调用者，如果不是语法限制，其实是可以很方便得到调用者的所有数据。

分类: 解惑

标签: c, 汇编, 内存, 编译

琴鸟
关注 - 1
粉丝 - 2
+加关注

« 上一篇：sqlcmd 登录和执行语句。
» 下一篇：c 函数调用产生的汇编指令和数据在内存情况(2)

posted @ 2016-05-08 15:56 琴鸟 阅读(69) 评论(0) 编辑 收藏

» 更多新闻...

最新知识库文章:

· 循序渐进地代码重构
· 技术的正宗与野路子
· 陈皓：什么是工程师文化？
· 没那么难，谈CSS的设计模式
· 程序猿媳妇儿注意事项
» 更多知识库文章...