X64调用约定

原创

qq_857305819 于 2020-09-29 10:49:55 发布 531 收藏 4

分类专栏： X64

版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。

本文链接：https://blog.csdn.net/qq_41490873/article/details/108861178

版权

X64 专栏收录该内容

15 篇文章 0 订阅

订阅专栏

fastcall

在64位下只有一种调用约定，类似于fastcall，要写其他的也不会出错，这主要是为了兼容以前定义的32的头文件。

前4个参数使用 rcx rdx r8 r9传递

写一个没有参数没有返回值的函数反汇编如下，可以看出不再像32位一样push ebp mov ebp,esp.
而是只保存了rdi就推出了。

```
void func()
{
00007FF6E1F61020  push        rdi

}
00007FF6E1F61022  pop         rdi
00007FF6E1F61023  ret
```

1个参数

一个参数的情况下采用ecx传参

```
00007FF695E51043  rep stos    dword ptr [rdi]
    func(3);
00007FF695E51045  mov         ecx,3
00007FF695E5104A  call        func (07FF695E5100Fh)
    getchar();
```

参数大于32位采用rcx传参

```
00007FF68B37103E  mov         eax,0CCCCCCCCh
00007FF68B371043  rep stos    dword ptr [rdi]
    func(0x123456789);
00007FF68B371045  mov         rcx,123456789h
00007FF68B37104F  call        func (07FF68B371005h)
    getchar();
```

函数里面把参数复制到了rsp+8

```
void func(int a)
{
00007FF695E51020  mov         dword ptr [rsp+8],ecx
00007FF695E51024  push        rdi

}
00007FF695E51025  pop         rdi
```

4个参数的情况

参数采用ecx,edx,r8d,r9d

```
    func(1, 2, 3, 4);
00007FF6FE577DE5  mov         r9d, 4
00007FF6FE577DEB  mov         r8d, 3
00007FF6FE577DF1  mov         edx, 2
00007FF6FE577DF6  mov         ecx, 1
00007FF6FE577DFB  call        func (07FF6FE56100Fh)
    getchar();
```

参数大于32位采用的是rcx,rdx,r8,r9寄存器

```
    func(0x123456789, 0x123456789, 0x123456789, 0x123456789);
00007FF69B737DE5  mov         r9, 123456789h
00007FF69B737DEF  mov         r8, 123456789h
00007FF69B737DF9  mov         rdx, 123456789h
00007FF69B737E03  mov         rcx, 123456789h
00007FF69B737E0D  call        00007FF69B721014
    getchar();
```

在函数里，把参数复制到rsp+0x20 +0x18 +0x10 +0x8的位置

```
void func(int a, int b, int c, int d)
{
00007FF6FE577DB0  mov         dword ptr [rsp+20h], r9d
00007FF6FE577DB5  mov         dword ptr [rsp+18h], r8d
00007FF6FE577DBA  mov         dword ptr [rsp+10h], edx
00007FF6FE577DBE  mov         dword ptr [rsp+8], ecx
00007FF6FE577DC2  push        rdi

}
```

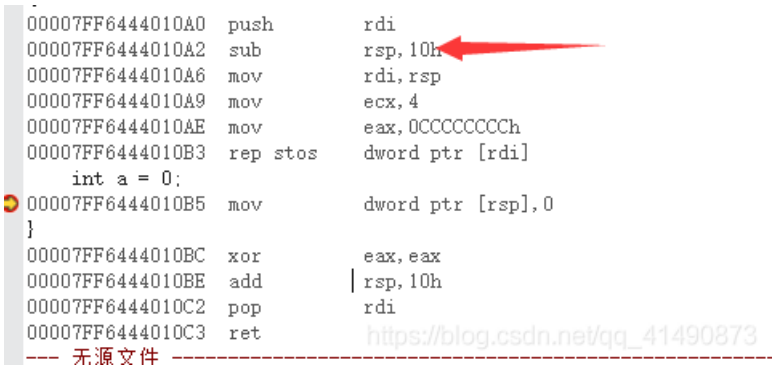
大于4个参数的情况

传参多了一个rsp+0x20内存.加上call对esp+8的位置，实际在函数里这个内存地址是rsp+0x28。

```
    func(0x123456789, 0x123456789, 0x123456789, 0x123456789, 0x123456789);
00007FF7B73C7DE5  mov         rax, 123456789h
00007FF7B73C7DEF  mov         qword ptr [rsp+20h], rax
00007FF7B73C7DF4  mov         r9, 123456789h
00007FF7B73C7DFE  mov         r8, 123456789h
00007FF7B73C7E08  mov         rdx, 123456789h
00007FF7B73C7E12  mov         rcx, 123456789h
00007FF7B73C7E1C  call        00007FF7B73B1019
    getchar();
00007FF7B73C7E21  call        00007FF7B73B1084
```

寄存器传参也要分配栈

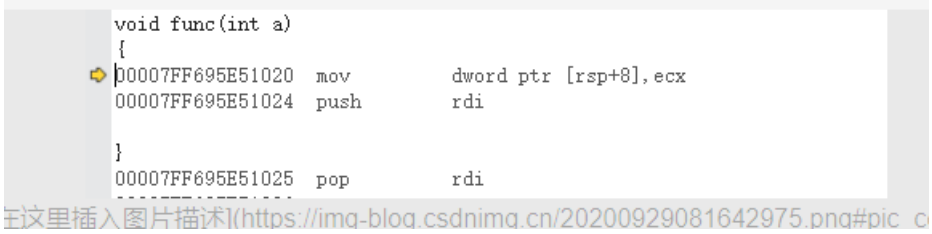
首先看一下没有调用函数时提升的栈空间是0x10



调用一个没有参数的函数，栈空间提升了0x10. 这是给函数的返回地址和多给了8字节的栈空间。
这8字节是给函数内部保存寄存器参数的值的。mov [rsp+8],rcx

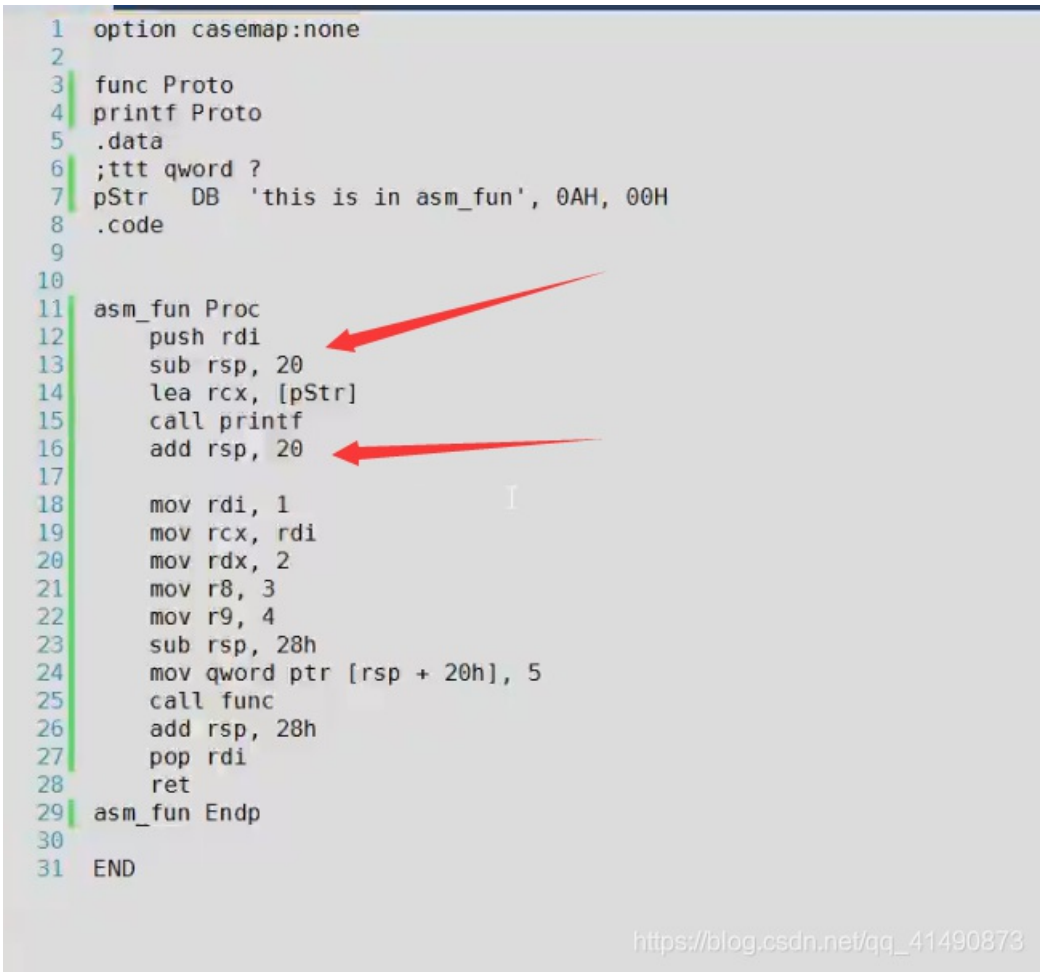
易变寄存器和非异变寄存器操作

异变寄存器：rax rcx rdx r8 r9 r10 r11 其余为非异变寄存器
push pop指令仅用来保存非易变寄存器。
意思是如果要在自己的汇编代码里使用非异变寄存器需要先保存
从一个空的函数里面就可以看出来。



汇编调用函数的问题

即使函数只有一个参数，也得分配0x20的栈空间 sub rsp,0x20



```
option casemap:none

func Proto
printf Proto
.data
;ttt qword ?
pStr   DB  'this is in asm_fun', 0AH, 00H
.code


asm_fun Proc
    push rdi
    sub rsp, 20
    lea rcx, [pStr]
    call printf
    add rsp, 20

    mov rdi, 1
    mov rcx, rdi
    mov rdx, 2
    mov r8, 3
    mov r9, 4
    sub rsp, 28h
    mov qword ptr [rsp + 20h], 5
    call func
    add rsp, 28h
    pop rdi
    ret
asm_fun Endp

END
```

https://blog.csdn.net/qq_41490873

通常不使用rbp寻址栈内存，所以rsp在函数中尽量保持稳定（一次性分配参数和变量空间）

如下图：连续调用4次函数，并没有像32位那样add esp，xxx

而是直接在函数main函数头部sub rsp,0x20 ，在尾部add sup,0x20

对于调用的函数参数少于4个字节的情况下栈空间就够了。多余4个会分配更多空间。

```
void main()
{
00007FF7CCC91040  push        rdi
00007FF7CCC91042  sub         rsp,20h
00007FF7CCC91046  mov         rdi,rsp
00007FF7CCC91049  mov         ecx,8
00007FF7CCC9104E  mov         eax,0CCCCCCCCh
00007FF7CCC91053  rep stos    dword ptr [rdi]
    test();
00007FF7CCC91055  call        test (07FF7CCC91005h)
    test();
00007FF7CCC9105A  call        test (07FF7CCC91005h)
    test();
00007FF7CCC9105F  call        test (07FF7CCC91005h)
    getchar();
00007FF7CCC91064  call        getchar (07FF7CCCAB190h)
}
```

https://blog.csdn.net/qq_41490873

像下面这样其中有一个函数的参数是5个会出现什么情况呢？

可以看出现在是sub rsp,0x30 也就是说编译器并不是按照最多参数5个分配0x28而是对其了0x10。

直接分配的0x30字节

```
void main()
{
00007FF67D26B1E0  push        rdi
00007FF67D26B1E2  sub         rsp,30h
00007FF67D26B1E6  mov         rdi,rsp
00007FF67D26B1E9  mov         ecx,0Ch
00007FF67D26B1EE  mov         eax,0CCCCCCCCh
00007FF67D26B1F3  rep stos    dword ptr [rdi]
    test();
00007FF67D26B1F5  call        test (07FF67D251005h)
    test();
00007FF67D26B1FA  call        test (07FF67D251005h)
    test();
00007FF67D26B1FF  call        test (07FF67D251005h)
    test1(1,2,3,4,5);
00007FF67D26B204  mov         dword ptr [rsp+20h],5
00007FF67D26B20C  mov         r9d,4
00007FF67D26B212  mov         r8d,3
00007FF67D26B218  mov         edx,2
00007FF67D26B21D  mov         ecx,1
00007FF67D26B222  call        test1 (07FF67D251014h)
    getchar();
00007FF67D26B227  call        getchar (07FF67D26B190h)
}
00007FF67D26B22C  xor         eax,eax
```

https://blog.csdn.net/qq_41490873