Windows平台X64函数调用约定与汇编代码分析

转载

BMOP 于 2014-09-09 14:17:02 发布 2138 收藏 4
分类专栏： 汇编语言


汇编语言 专栏收录该内容
7 篇文章 0 订阅
订阅专栏

原文 http://kelvinh.github.io/blog/2013/08/05/windows-x64-calling-conventions/

起因

整件事源自于公司的一个公共模块，有很多项目都依赖于这个公共模块，我们项目是其中之一。假定依赖的函数原型为：

```
int add(int a, int b, int c, int d, int e)
```

某一天，这个公共模块将 `add` 函数增加了一个参数，即原型变成了下面这样：

```
int add(int a, int b, int c, int d, int e, int f)
```

但是公共模块的负责人没有通知我们，所以我们并不知道这个变化。

如果是常规调用，即直接用 `add(...)` 来调用的话，在公共模块增加参数后，那么编译就会提示出错的。但是我们是采用的动态调用，如下：

```
typedef int (*FunctionPtr)(int a, int b, int c, int d, int e);

HMODULE h = ::LoadLibraryEx(L"CommonModule.dll", ...);
FunctionPtr pf = reinterpret_cast<FunctionPtr>(::GetProcAddress(h, "add"));
int ret = pf(1, 2, 3, 4, 5);
//...
```

因此，对这种调用方式来说，在编译阶段，调用者是无法知晓 `CommonModule.dll` 的改动的。

可是，按照常规讲，虽然编译阶段没有出错，那么在运行的时候，总会发生异常，例如crash之类的啊。可奇怪的就在这一点，运行时也一切正常，QA测试也没发现什么bug，于是项目就正常release了。

但在release之后，另一个使用这个公共模块的项目出了问题，具体什么问题我不太清楚，后来检查了之后才发现是公共模块的接口有变化。于是，我们项目的老大们对这个公共模块的人大发牢骚：改接口也不通知，现在产品都release了，如果因为这个原因而大面积出问题，你怎么跟客户解释！！

可是，我们QA测试的时候不是一点问题都没有么，到底是怎么回事呢？

分析

于是，我写了如下的一个例子，来分析一番：

```
#include <iostream>

int add(int a, int b, int c, int d, int e) {
    return a + b + c + d + e;
}

int main() {
    int r = add(1, 2, 3, 4, 5);
    std::cin.get();
}
```

上面例子中 main() 函数调用 std::cin.get() 是为了让程序阻塞在那里，方便我们能够来得及在程序退出之前将WinDbg给Attach到进程上去。下面的就是在WinDbg里面看到的 main() 函数和 add() 函数的汇编代码（能看到如下代码的前提是，在Visual Studio编译之前，把相关的优化选项以及安全检查选项给关掉，不然Visual Studio会插入一些安全检查的代码，同时会直接把 add() 函数给优化掉）（忘了说明，编译采用的是VS2012）：

```
0:000> uf Project1!main
Project1!main [d:\codelabs\functest\c++\project1\source.cpp @ 7]:
    7 00000001`3ff21730 4883ec48        sub     rsp,48h
    8 00000001`3ff21734 c744242005000000 mov     dword ptr [rsp+20h],5
    8 00000001`3ff2173c 41b904000000    mov     r9d,4
    8 00000001`3ff21742 41b803000000    mov     r8d,3
    8 00000001`3ff21748 ba02000000      mov     edx,2
    8 00000001`3ff2174d b901000000      mov     ecx,1
    8 00000001`3ff21752 e8a9ffffff      call    Project1!add (00000001`3ff21700)
    8 00000001`3ff21757 89442430        mov     dword ptr [rsp+30h],eax
    9 00000001`3ff2175b 488b0dee180000  mov     rcx,qword ptr [Project1!_imp_?cinstd (00000001`3ff23050)]
    9 00000001`3ff21762 ff15f8180000    call    qword ptr [Project1!_imp_?get?$basic_istreamDU?$char_traits
sDstdstdQEAAHXZ (00000001`3ff23060)]
   10 00000001`3ff21768 33c0            xor     eax,eax
   10 00000001`3ff2176a 4883c448        add     rsp,48h
   10 00000001`3ff2176e c3              ret
0:000> uf Project1!add
Project1!add [d:\codelabs\functest\c++\project1\source.cpp @ 3]:
    3 00000001`3ff21700 44894c2420      mov     dword ptr [rsp+20h],r9d
    3 00000001`3ff21705 4489442418      mov     dword ptr [rsp+18h],r8d
    3 00000001`3ff2170a 89542410        mov     dword ptr [rsp+10h],edx
    3 00000001`3ff2170e 894c2408        mov     dword ptr [rsp+8],ecx
    4 00000001`3ff21712 8b442410        mov     eax,dword ptr [rsp+10h]
    4 00000001`3ff21716 8b4c2408        mov     ecx,dword ptr [rsp+8]
    4 00000001`3ff2171a 03c8            add     ecx,eax
    4 00000001`3ff2171c 8bc1            mov     eax,ecx
    4 00000001`3ff2171e 03442418        add     eax,dword ptr [rsp+18h]
    4 00000001`3ff21722 03442420        add     eax,dword ptr [rsp+20h]
    4 00000001`3ff21726 03442428        add     eax,dword ptr [rsp+28h]
    5 00000001`3ff2172a c3              ret
```

如果对于x64的CPU结构不太熟悉的话，上面的汇编着实不太好理解，下面给出一张x64结构的CPU寄存器构造图[1]：

从图上可以看到，X64架构相对于X32架构的主要变化，是将原来所有的寄存器都扩大了一倍，例如EAX现在扩充成RAX，同时，又新增加了从R8到R15这8个64位的寄位器。

知道了寄存器的结构，还是不足以理解上面的汇编函数调用，此外，还需要知道Intel X64汇编函数调用的一些约定[2]：

> RCX, RDX, R8, R9 are used for integer and pointer arguments in that order left to right.
>
> XMM0, 1, 2, and 3 are used for floating point arguments.
>
> Additional arguments are pushed on the stack left to right.
>
> Parameters less than 64 bits long are not zero extended; the high bits contain garbage.
>
> It is the caller's responsibility to allocate 32 bytes of "shadow space" (for storing RCX, RDX, R8, and R9 if needed) before calling the function.
>
> It is the caller's responsibility to clean the stack after the call.
>
> Integer return values (similar to x86) are returned in RAX if 64 bits or less.
>
> Floating point return values are returned in XMM0.
>
> Larger return values (structs) have space allocated on the stack by the caller, and RCX then contains a pointer to the return space when the callee is called. Register usage for integer parameters is then pushed one to the right. RAX returns this address to the caller.
>
> The stack is 16-byte aligned. The "call" instruction pushes an 8-byte return value, so the all non-leaf functions must adjust the stack by a value of the form 16n+8 when allocating stack space.
>
> Registers RAX, RCX, RDX, R8, R9, R10, and R11 are considered volatile and must be considered destroyed on function calls.
>
> RBX, RBP, RDI, RSI, R12, R14, R14, and R15 must be saved in any function using them.
>
> Note there is no calling convention for the floating point (and thus MMX) registers.
>
> Further details (varargs, exception handling, stack unwinding) are at Microsoft's site.

上面这段话里面有几个关键点：1．一个函数在调用时，前四个参数是从左至右依次存放于RCX、RDX、R8、R9寄存器里面；2．剩下的参数从左至右顺序入栈；3．调用者负责在栈上分配32字节的“shadow space”，用于存放那四个存放调用参数的寄存器的值（亦即前四个调用参数）；4．调用者负责清理栈；5．被调用函数若是整数返回，则返回值会被存放于RAX；6．栈是16字节对齐的，“call”指令会入栈一个8字节的返回值（注：即函数调用前原来的RIP指令寄存器的值），这样一来，栈就对不齐了，所以，所有非叶子结点调用的函数，都必须调整栈分配方式为16n+8，来使栈对齐。

这样一来，上面的汇编代码就好懂了：

> 主调函数（main）将栈指针RSP下移0x48，即分配栈空间；
>
> 将最后一个调用参数5入栈，存放于[RSP + 0x20]处，这样一来，栈上面空出的 0x48 - 0x20 = 0x28 = 40 = 32 + 8 的空间就用于存放本地变量，其中8字节应该是用来对齐栈的；
>
> 将前四个参数分别放入约定中的那四个寄存器；
>
> 调用 add 函数（在这个指令中，栈指针RSP又下移了8个字节，这8个字节用来存放RIP指针的值）；
>
> 进入 add 函数，把在四个寄存器中的参数又放回栈上（栈上的用于存放这四个寄存器空间就是“shadow space”，如果需要，由被调用者负责将这四个寄存器的值放回栈[3]），然后执行加操作，最后的结果存放于RAX中；
>
> 返回 main 函数后，取出RAX的值，再放回本地变量的栈空间中；

```
...
```

上面的指令对栈操作比较多，我画了一个调用栈的分配情况图（用Window Paint画的，花的时间不比写这篇博客的时间短，中间还画错了一次。。 =_=#!）：

另外，再附上一张MSDN上画的栈分配的示例图[4]（和上面我画的差不多，只不过我画的是针对于具体的例子）：

结论

按照分析中所指出的，在X64平台的函数调用中，函数前四个参数之外的参数会被入栈，如上面的栈分配图所示，实际上在主调函数中，分配的栈参数空间上面还有本地变量空间，所以，如果偶然多了一个参数的话，其实没有关系，不会导致crash，只不过这个参数的值会是无效的。既然参数值无效，程序运行应该会出bug才对，后来问了这个公共模块的负责人，才知道这个新加的参数没有使用。。擦，没有使用，你乱改接口，新加参数干嘛。。