Linux Debugging（三）：C++函數調用的參數傳遞方法總結（通過gdb+反彙編）

By anzhsoft | Published 2014年1月25日

上一篇文章《Linux Debugging:使用反彙編理解C++程序函數調用棧》沒想到能得到那麼多人的喜愛，因為那篇文章是以32位的C++普通函數（非類成員函數）為例子寫的，因此只是一個特殊的例子。本文將函數調用時的參數傳遞方法進行一下總結。總結將為C++普通函數、類成員函數；32位和64位進行總結。

建議還是讀一下Linux Debugging:使用反彙編理解C++程序函數調用棧，這樣本文的結論將非常容易理解，將非常好的為CoreDump分析開一個好頭。而且，它也是32位C++普通函數的調用的比較好的例子，畢竟從彙編的角度，將參數如何傳遞的進行了比較好的說明。

1. 32位程序普通函數

普通函數的意思是非class member function

```
void func2(int a, int b)
{
  a++;
  b+ = 2;
}

int main()
{
  func2( 1111, 2222);
  return 0;
}
```

main函數的彙編：

```
main:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        movl $2222, 4(%esp)
        movl $1111, (%esp)
        call func2(int, int)
        movl $0, %eax
        leave
        ret
```

1111是第一個參數，放到了esp指向的地址。2222是第二個參數，放到了高地址。因次我們可以知道，在函數func2中，通過ebp+8可以訪問到第一個參數1111，通過ebp+12可以訪問到第二個參數2222。

```
func2(int, int):
            pushl %ebp
            movl %esp, %ebp
            addl $1, 8(%ebp)
            addl $2, 12(%ebp)
            popl %ebp
            ret
```

下面我們使用gdb通過ebp打印一下傳入的參數：

```
anzhsoft@ubuntu:~/linuxDebugging/parameter$ gdb a.out
GNU gdb 6.8
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
```

and "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) b func2
Breakpoint 1 at 0x8048597: file m32noclass.cpp, line 6.
(gdb) r
Starting program: /home/anzhsoft/linuxDebugging/parameter/a.out

Breakpoint 1, func2 (a=1111, b=2222) at m32noclass.cpp:6
warning: Source file is more recent than executable.
6 a++;
(gdb) p *(int*)($ebp+8)
$1 = 1111
(gdb) p *(int*)($ebp+12)
$2 = 2222

總結：

1. 參數通過棧查傳遞，底地址傳遞從左邊開始的第一個參數

2. 使用gdb可以很方便打印傳入參數

(gdb) p *(int*)($ebp+8)
$1 = 1111
(gdb) p *(int*)($ebp+12)
$2 = 2222

其實32位的程序也可以使用寄存器傳遞參數。請看下一節。

2. 32位普通函數-通過寄存器傳遞參數

可以使用GCC的擴展功能__attribute__使得參數傳遞可以使用寄存器。

修改第一節的函數：

```
#define STACKCALL __attribute__((regparm(3)))
void STACKCALL func4(int a, int b, int c, int d)
{
  a++;
  b += 2;
  c += 3;
  d += 4;
}

int main()
{
  func4(1111, 2222, 3333, 4444);
  return 0;
}
```

__attribute__((regparm(3)))意思是使用寄存器

```
anzhsoft@ubuntu:~/linuxDebugging/parameter$ gdb a.out
GNU gdb 6.8
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) b main
Breakpoint 1 at 0x80485bb: file m32noclass.cpp, line 14.
```

```
(gdb) r
Starting program: /home/anzhsoft/linuxDebugging/parameter/a.out

Breakpoint 1, main () at m32noclass.cpp:14
14 func4(1111, 2222, 3333, 4444);
(gdb) s
func4 (a=1111, b=2222, c=3333, d=4444) at m32noclass.cpp:6
6 a++;
(gdb) ir
eax 0x457 1111
ecx 0xd05 3333
edx 0x8ae 2222
ebx 0xb3eff4 11792372
esp 0xbf8e9580 0xbf8e9580
ebp 0xbf8e958c 0xbf8e958c
esi 0x0 0
edi 0x0 0
eip 0x80485a3 0x80485a3 <func4(int, int, int, int)+15>
eflags 0x282 [ SF IF ]
cs 0x73 115
ss 0x7b 123
ds 0x7b 123
es 0x7b 123
fs 0x0 0
gs 0x33 51
(gdb) p *(int*)($ebp+8)
$1 = 4444
(gdb)
```

可以看到，前三個參數分別通過eax/edx/ecx**傳遞，第四個參數通過棧傳遞。這種傳遞方式被稱為**fastcall

3. 32位class member function 參數傳遞方式

我們通過宏USINGSTACK強制使用棧來傳遞參數。

```
#define USINGSTACK __attribute__((regparm(0)))
class Test
{
public:
  Test():number(3333){}
  void USINGSTACK func2(int a, int b)
  {
    a++;
    b += 2;
  }
private:
  int number;
};

int main(int argc, char* argv[])
{
  Test tInst;
  tInst.func2(1111, 2222);
  return 0;
}
```

通過gdb來打印傳入的參數：

```
anzhsoft@ubuntu:~/linuxDebugging/parameter$ gdb a.out
GNU gdb 6.8
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
```

```
and "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) b main
Breakpoint 1 at 0x804859d: file m32class.cpp, line 18.
(gdb) r
Starting program: /home/anzhsoft/linuxDebugging/parameter/a.out

Breakpoint 1, main (argc=1, argv=0xbf98a454) at m32class.cpp:18
18 Test tInst;
(gdb) n
19 tInst.func2(1111, 2222);
(gdb) s
Test::func2 (this=0xbf98a39c, a=1111, b=2222) at m32class.cpp:9
9 a++;
(gdb) p *(int*)($ebp+12)
$1 = 1111
(gdb) p *(int*)($ebp+16)
$2 = 2222
(gdb) p *this
$3 = {number = 3333}
```

可以看到，class成員函數的第一個參數是對象的指針。通過這種方式，成員函數可以和非成員函數以類似的方式進行調用。

4. x86-64 class member function 的參數傳遞

在x86-64中，整形和指針型參數的參數從左到右依次保存到rdi，rsi，rdx，rcx，r8，r9中。浮點型參數會保存到xmm0，xmm1……。多餘的參數會保持到棧上。

下面這個例子將傳遞九個參數。可以通過它來驗證一下各個寄存器的使用情況：

```
class Test
{
```

```
public:
  Test():number(5555){}
  void func9(int a, int b, int c, int d,char*str, long e, long f, float h, double i)
  {
    a++;
    b += 2;
    c += 3;
    d += 4;
  }
private:
  int number;
};

int main(int argc, char* argv[])
{
  Test tInst;
  tInst.func9(1111, 2222, 3333, 4444, "hello, world!", 6666,7777, 8.888, 9.999);
  return 0;
}
```

下面通過gdb驗證各個寄存器的使用情況：

```
khawk-dev-zhanga12:~/study/c++callstack # gdb a.out
GNU gdb (GDB) SUSE (7.0-0.4.16)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-suse-linux".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /root/study/c++callstack/a.out...done.
(gdb) b main
Breakpoint 1 at 0x40071b: file m64class.cpp, line 19.
(gdb) r
```

```
Starting program: /root/study/c++callstack/a.out
Breakpoint 1, main (argc=1, argv=0x7fffffffe188) at m64class.cpp:19
19 Test tInst;
(gdb) n
20 tInst.func9(1111, 2222, 3333, 4444, "hello, world!", 6666,7777, 8.888, 9.999);
(gdb) s
Test::func9 (this=0x7fffffffe0a0, a=1111, b=2222, c=3333, d=4444, str=0x400918 "hello,
world!", e=6666, f=7777, h=8.88799953, i=9.9990000000000006 )
    at m64class.cpp:8
8 a++;
(gdb) info reg
rax 0x400918 4196632
rbx 0x400830 4196400
rcx 0xd05 3333
rdx 0x8ae 2222
rsi 0x457 1111
rdi 0x7fffffffe0a0 140737488347296
rbp 0x7fffffffe070 0x7fffffffe070
rsp 0x7fffffffe070 0x7fffffffe070
r8 0x115c 4444
r9 0x400918 4196632
r10 0xffffffffffffffff -1
r11 0x7ffff733d890 140737340758160
r12 0x400620 4195872
r13 0x7fffffffe180 140737488347520
r14 0x0 0
r15 0x0 0
rip 0x4007ff 0x4007ff <Test::func9(int, int, int, int, char*, long, long, float, double)+35>
eflags 0x202 [ IF ]
cs 0x33 51
ss 0x2b 43
ds 0x0 0
es 0x0 0
fs 0x0 0
gs 0x0 0
fctrl 0x37f 895
fstat 0x0 0
```

```
ftag 0xffff 65535
fiseg 0x0 0
fioff 0x0 0
foseg 0x0 0
fooff 0x0 0
fop 0x0 0
mxcsr 0x1f80 [ IM DM ZM OM UM PM ]
(gdb) p *this
$1 = {number = 5555}
(gdb) p *(char*)$r9@13
$2 = "hello, world!"
(gdb) p *(int*)($rbp+16)
$4 = 6666
(gdb) p *(int*)($rbp+24)
$5 = 7777
(gdb) p *(int*)$rdi
$6 = 5555
```

r9存儲的是指針型char *str的字符串。因為寄存器只能存儲6個整形、指針型參數，注意，Test對象的指針佔用了一個。因此

參數e和f只能通過棧傳遞。注意每個地址空間佔8個字節。因此rbp+2*8存儲的是e，rbp+3*8存儲的是f。

float h是通過xmm0，double i是通過xmm1傳遞的。這類寄存器的size大小是128bits，當然128個bits可以不填滿。GDB將這些寄存器看成下面這些數據的聯合：

```
union{
  float v4_float[4];
  double v2_double[2];
  int8_t v16_int8[16];
  int16_t v8_int16[8];
  int32_t v4_int32[4];
  int64_t v2_int64[2];
```

```
  int128_t unit128;
}xmm0,xmm1,xmm2,xmm3,xmm4,xmm5,xmm6,xmm7;
```

打印方式如下：

```
(gdb) p $xmm0.v4_float[0]
$7 = 8.88799953
(gdb) p $xmm1.v2_double[0]
$8 = 9.9990000000000006
```

總結：

在x86-64中，整形和指針型參數的參數從左到右依次保存到rdi，rsi，rdx，rcx，r8，r9中。浮點型參數會保存到xmm0，xmm1……。多餘的參數會保持到棧上。

5. 總結

32位：

1）默認的傳遞方法不使用寄存器，使用ebp+8可以訪問第一個參數，ebp+16可以訪問第二個參數。。。

2） 可以使用GCC擴展__attribute__將參數放到寄存器上，依次使用的寄存器是eax/edx/ecx

3）對於class 的member function，調用時第一個參數為this（對象指針）地址。因此函數的第一個參數使用ebp+16才可以訪問到。

4）ebp存儲的是上層的bp的地址。ebp+4存儲的是函數返回時的地址指令。

64位：

1）默認的傳遞方法是使用寄存器。當然也可以強制使用棧，方式：函數聲明時使用

__attribute__((regparm(0)))

2）整形和指針型參數的參數從左到右依次保存到rdi，rsi，rdx，rcx，r8，r9中。浮點型參數會保存到xmm0，xmm1……。多餘的參數會保持到棧上。

3）xmm0……是比較特殊的寄存器，訪問內容時需要注意。