Linux Debugging（一）：使用反彙編理解C++程序函數調用棧

By anzhsoft | Published 2014年1月24日

拿到CoreDump後，如果看到的地址都是？？？？，那麼基本上可以確定，程序的棧被破壞掉了。GDB也是使用函數的調用棧去還原“事故現場”的。因此理解函數調用棧，是使用GDB進行現場調試或者事後調試的基礎，如果不理解調用棧，基本上也從GDB得不到什麼有用的信息。當然了，也有可能你非常“幸運”， 一個bt就把哪兒越界給標出來了。但是，大多數的時候你不夠幸運，通過log，通過簡單的code walkthrough，得不到哪兒出的問題；或者説只是推測，不能確診。我們需要通過GDB來最終確定CoreDump產生的真正原因。

本文還可以幫助你深入理解C++函數的局部變量。我們學習時知道局部變量是是存儲到棧裡的，內存管理對程序員是透明的。通過本文，你將明白這些結論是如何得出的。

棧，是LIFO（Last In First Out）的數據結構。C++的函數調用就是通過棧來傳遞參數，保存函數返回後下一步的執行地址。接下來我們通過一個具體的例子來探究。

```
int func1(int a)
{
  int b = a + 1;
  return b;
}
int func0(int a)
{
  int b = func1(a);
  return b;
}

int main()
{
  int a = 1234;
  func0(a);
  return 0;
```

}

可以使用以下命令將上述code編程成彙編代碼：

g++ -g -S -O0 -m32 main.cpp -o-|c++filt >main.format.s

c++filt 是為了Demangle symbols。-m32是為了編譯成x86-32的。因為對於x86-64來說，函數的參數是通過寄存器傳遞的。

main的彙編代碼：

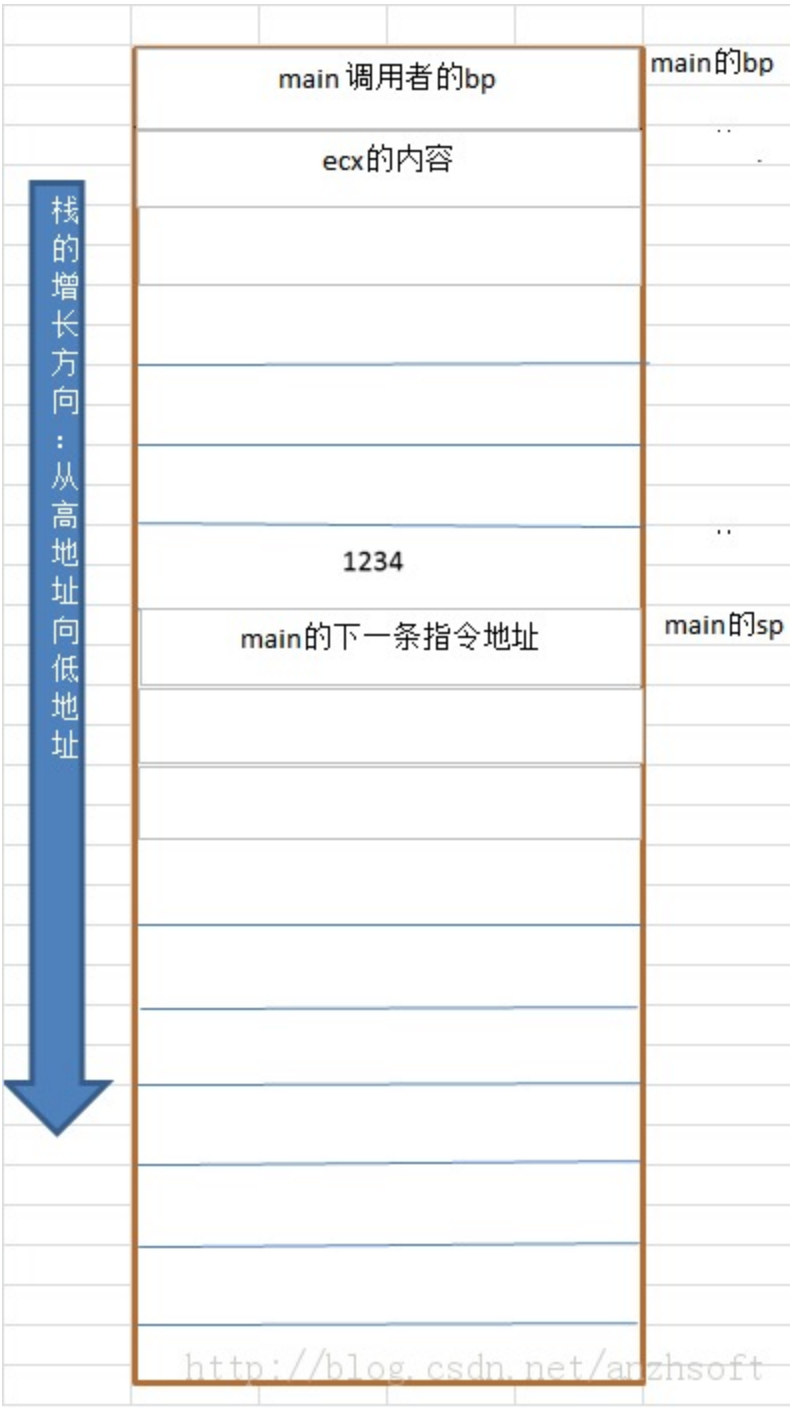
```
main:
        leal 4(%esp), %ecx
        andl $-16, %esp
        pushl -4(%ecx)

        pushl %ebp #1：push %ebp指令把ebp寄存器的值壓棧，同時把esp的值減4
        movl %esp, %ebp #2 把esp的值傳送給ebp寄存器。
                        #1 + #2 合起來是把原來ebp的值保存在棧上，然後又給ebp賦了新值。
                        #2+ ebp指向棧底，而esp指向棧頂，在函數執行過程中esp
                        #2++隨著壓棧和出棧操作隨時變化，而ebp是不動的
        pushl %ecx
        subl $20, %esp #3 現在esp地址-20/4 = 5, 及留出5個地址空間給main的局部變量
        movl $1234, -8(%ebp)#4 局部變量1234 存入ebp - 8 的地址
        movl -8(%ebp), %eax #5 將地址存入eax
        movl %eax, (%esp) #6 將1234存入esp指向的地址
        call func0(int) #7 調用func0，注意這是demangle後的函數名，實際是一個地址
        movl $0, %eax
        addl $20, %esp
        popl %ecx
        popl %ebp
        leal -4(%ecx), %esp
        ret
```

對於call指令，這個指令有兩個作用：

1. `func0`函數調用完之後要返回到`call`的下一條指令繼續執行，所以把`call`的下一條指令的地址壓棧，同時把`esp`的值減4。
2. 修改程序計數器`eip`，跳轉到`func0`函數的開頭執行。

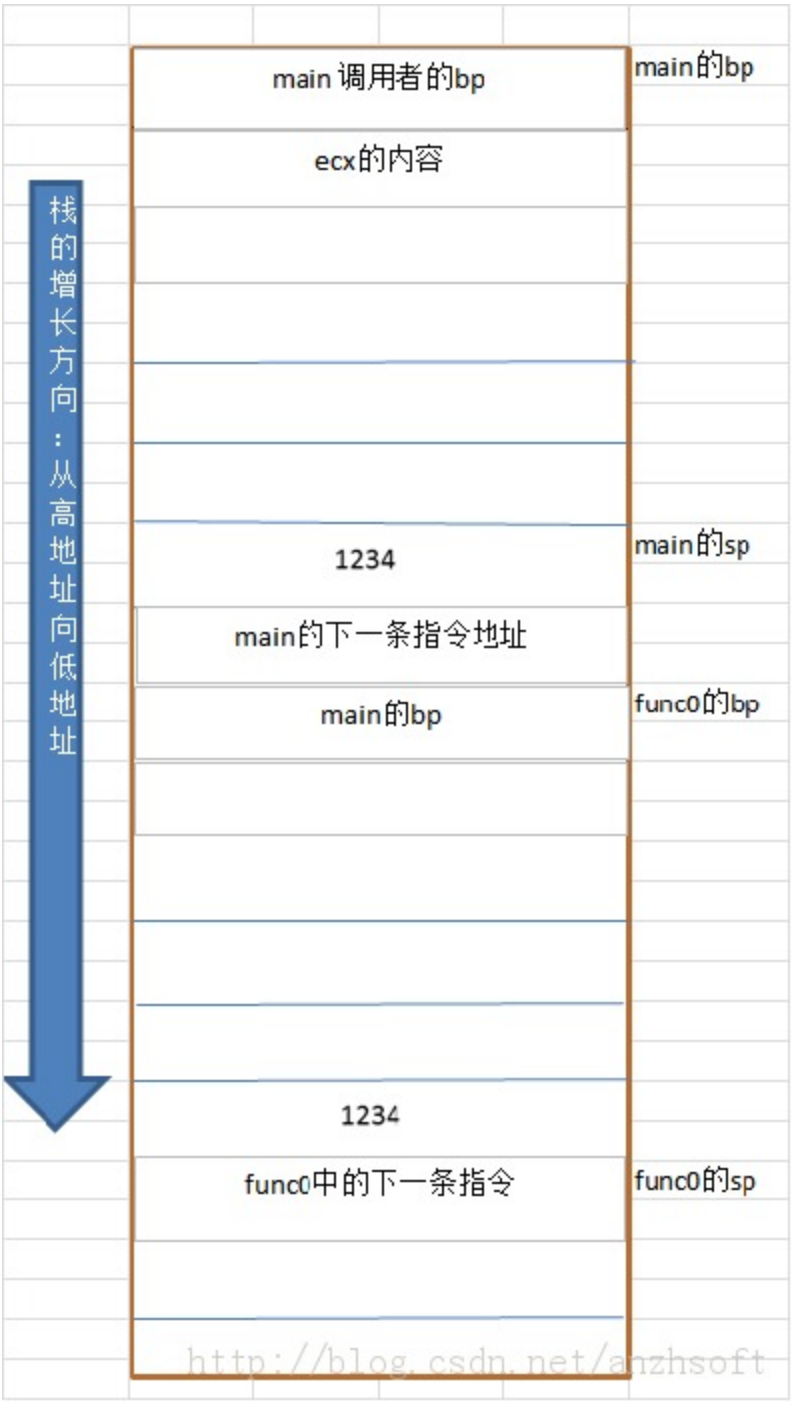
至此，調用funco的棧就是下面這個樣子：


main 调用者的bp
main的bp
ecx的内容
栈的增长方向：从高地址向低地址
1234
main的下一条指令地址
main的sp
http://blog.csdn.net/anzhsoft

下面看一下func0的彙編代碼：

```
func0(int):
        pushl %ebp
        movl %esp, %ebp
        subl $20, %esp
        movl 8(%ebp), %eax
        movl %eax, (%esp)
        call func1(int)
        movl %eax, -4(%ebp)
        movl -4(%ebp), %eax
        leave
        ret
```

需要注意的是esp也是留了5個地址空間給func0使用。並且ebp的下一個地址就是留給局部變量b的，調用棧如圖：



通過調用栈可以看出，8(%ebp)**其實就是傳入的參數**1234**。**

func1的代碼：

```
func1(int):
        pushl %ebp
        movl %esp, %ebp
        subl $16, %esp
        movl 8(%ebp), %eax #去傳入的參數，即1234
        addl $1, %eax # +1 運算
        movl %eax, -4(%ebp)
        movl -4(%ebp), %eax #將計算結果存入eax，這就是返回值
        leave
        ret
```

`leave`指令，這個指令是函數開頭的`push %ebp`和`mov %esp,%ebp`的逆操作：

1. 把`ebp`的值賦給`esp`
2. 現在`esp`所指向的棧頂保存著`foo`函數棧幀的`ebp`，把這個值恢復給`ebp`，同時`esp`增加4。注意，現在esp指向的是這次調用的返回地址，即上次調用的下一條執行指令。

最後是`ret`指令，它是`call`指令的逆操作：

1. 現在`esp`所指向的棧頂保存著返回地址，把這個值恢復給`eip`，同時`esp`增加4，`esp`**指向了當前**`frame`**的棧頂**。
2. 修改了程序計數器`eip`，因此跳轉到返回地址繼續執行。

調用棧如下：

至此，func1返回後，控制權交還給func0，當前的棧就退化成func0的棧的情況，因為棧保存了一切信息，因此指令繼續執行。直至func0執行

leave

ret

以同樣的方式將控制權交回給main。

到這裡，你應該知道下面問題的答案了：

1. 局部變量的生命週期，

2. 局部變量是怎麼樣使用內存的；

3. 為什麼傳值不會改變原值（因為編譯器已經幫你做好拷貝了）

4. 為什麼會有棧溢出的錯誤

5. 為什麼有的寫壞棧的程序可以運行，而有的卻會crash（如果棧被破壞的是數據，那麼數據是臟的，不應該繼續運行；如果破壞的是上一層調用的bp，或者返回地址，那麼程序會crash，or unexpected behaviour...）

小節一下：

1. 在32位的機器上，C++的函數調用的參數是存到棧上的。當然gcc可以在函數聲明中添加__attribute__((regparm(3)))使用eax， edx，ecx傳遞開頭三個參數。

2. 通過bp可以訪問到調用的參數值。

3. 函數的返回地址（函數返回後的執行指令）也是存到棧上的，有目的的修改它可以使程序跳轉到它不應該的地方。。。

4. 如果程序破壞了上一層的bp的地址，或者程序的返回地址，那麼程序就很有可能crash

5. 拿到一個CoreDump，應該首先先看有可能出問題的線程的的frame的棧是否完整。

6. 64位的機器上，參數是通過寄存器傳遞的，當然寄存器不夠用就會通過棧來傳遞