函数调用过程探究

引言

如何定义函数、调用函数，是每个程序员学习编程的入门课。调用函数(caller)向被调函数(callee)传入参数，被调函数返回结果，看似简单的过程，其实CPU和系统内核在背后做了很多工作。下面我们通过反汇编工具，来看函数调用的底层实现。

基础知识

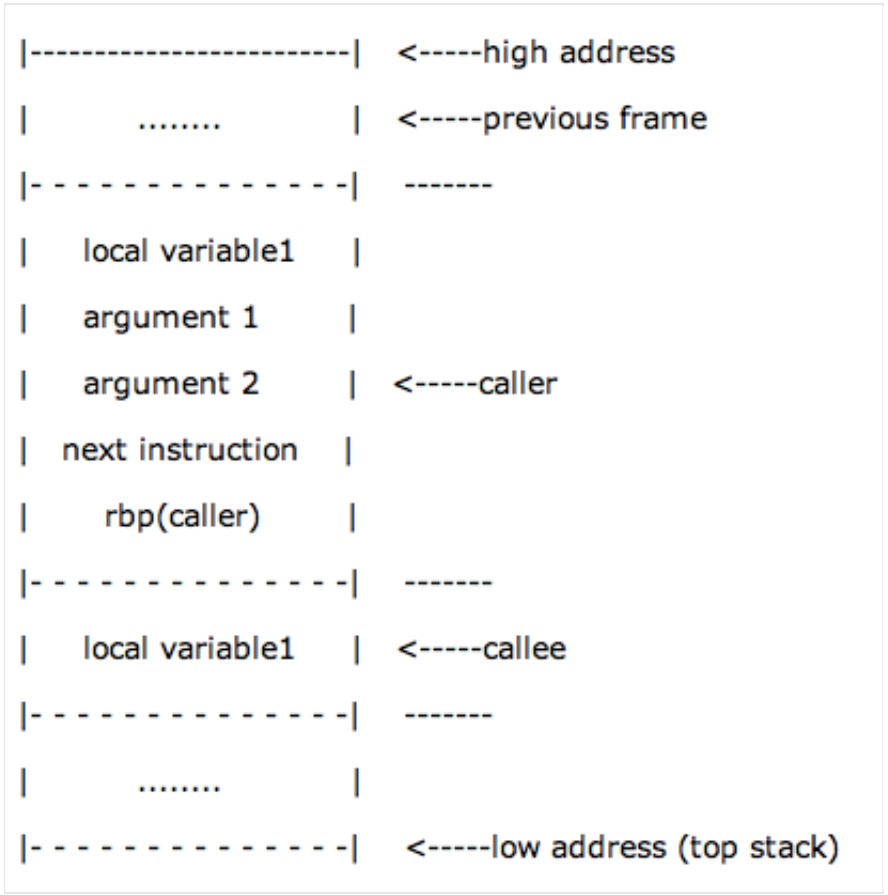
我们先来看几个概念，这有助于理解后面反汇编的输出结果。

栈(stack)

栈，相信大家都十分熟悉，push/pop，只允许在一端进行操作，后进先出(LIFO)，凡是学过编程的人都能列出一二三点。但就是这个最简单的数据结构，构成了计算机中程序执行的基础，用于内核中程序执行的栈具有以下特点：

- 每一个进程在用户态对应一个调用栈结构(call stack)
- 程序中每一个未完成运行的函数对应一个栈帧(stack frame)，栈帧中保存函数局部变量、传递给被调函数的参数等信息
- 栈底对应高地址，栈顶对应低地址，栈由内存高地址向低地址生长

一个进程的调用栈图示如下：



寄存器(register)

寄存器位于CPU内部，用于存放程序执行中用到的数据和指令，CPU从寄存器中取数据，相比从内存中取快得多。寄存器又分通用寄存器和特殊寄存器。

通用寄存器有ax/bx/cx/dx/di/si，尽管这些寄存器在大多数指令中可以任意选用，但也有一些规定某些指令

只能用某个特定“通用”寄存器，例如函数返回时需将返回值mov到ax寄存器中；特殊寄存器有bp/sp/ip等，特殊寄存器均有特定用途，例如sp寄存器用于存放以上提到的栈帧的栈顶地址，除此之外，不用于存放局部变量，或其他用途。

对于有特定用途的几个寄存器，简要介绍如下：

- **ax(accumulator)**: 可用于存放函数返回值
- **bp(base pointer)**: 用于存放执行中的函数对应的栈帧的栈底地址
- **sp(stack poinger)**: 用于存放执行中的函数对应的栈帧的栈顶地址
- **ip(instruction pointer)**: 指向当前执行指令的下一条指令

不同架构的CPU，寄存器名称被添以不同前缀以指示寄存器的大小。例如对于x86架构，字母“e”用作名称前缀，指示各寄存器大小为32位；对于x86_64寄存器，字母“r”用作名称前缀，指示各寄存器大小为64位。

函数调用例子

了解了栈和寄存器的概念，下面看一个函数调用实例：

```
//func_call.c
int bar(int c, int d)
{
    int e = c + d;
    return e;
```

```
}
int foo(int a, int b)
{
    return bar(a, b);
}
int main(void)
{
    foo(2, 5);
    return 0;
}
```

该程序很简单，main->foo->bar，编译得到可执行文件func_call：

```
# gcc -g func_call.c -o func_call
```

-g选项使目标文件func_call包含程序的调试信息。

反汇编分析

下面我们使用gdb对func_call进行反汇编，跟踪main->foo->bar函数调用过程。

```
# gdb func_call
//此处省略gdb版本信息
```

```
Reading symbols from /tmp/lx/func_call...done.
(gdb) start
Temporary breakpoint 1 at 0x400525: file func_call.c, line 14.
Starting program: /tmp/lx/func_call

Temporary breakpoint 1, main () at func_call.c:14
14              foo(2, 5);
(gdb)
```

start命令用于拉起被调试程序，并执行至main函数的开始位置，程序被执行之后与一个用户态的调用栈关联。

main函数

现进程跑在main函数中，我们disassemble命令显示当前函数的汇编信息：

```
(gdb) disassemble /rm
Dump of assembler code for function main:
13        {
0x0000000000400521 <main+0>:     55                  push %rbp
0x0000000000400522 <main+1>:     48 89 e5            mov %rsp,%rbp
```

```
14                  foo(2, 5);
0x0000000000400525 <main+4>:     be 05 00 00 00    mov $0x5,%esi
0x000000000040052a <main+9>:     bf 02 00 00 00    mov $0x2,%edi
0x000000000040052f <main+14>:    e8 d2 ff ff ff    callq 0x400506 <foo>

15                  return 0;
0x0000000000400534 <main+19>:    b8 00 00 00 00    mov $0x0,%eax

16          }
0x0000000000400539 <main+24>:     c9               leaveq
0x000000000040053a <main+25>:     c3               retq

End of assembler dump.
```

disassemble命令的/m指示显示汇编指令的同时，显示相应的程序源码；/r指示显示十六进制的计算机指令(raw instruction)。

以上输出每行指示一条汇编指令，除程序源码外共有四列，各列含义为:

1. **0x0000000000400521**: 该指令对应的虚拟内存地址
2. **<main+0>**: 该指令的虚拟内存地址偏移量
3. **55**: 该指令对应的计算机指令

4. **push %rbp**: 汇编指令

一个函数被调用，首先默认要完成以下动作：

- 将调用函数的栈帧栈底地址入栈，即将bp寄存器的值压入调用栈中
- 建立新的栈帧，将被调函数的栈帧栈底地址放入bp寄存器中

以下两条指令即完成上面动作：

```
push %rbp
mov  %rsp, %rbp
```

也许你会问：咦？以上disassemble的输出不是main函数的汇编指令吗，怎么输出中也有上面两条指令？难道main也是一个“被调函数”？

是的，皆因main并不是程序拉起后第一个被执行的函数，它被_start函数调用，更详细的资料参看这里。

一个函数调用另一个函数，需先将参数准备好。main调用foo函数，两个参数传入通用寄存器中：

```
mov $0x5, %esi
mov $0x2, %edi
```

对于参数传递的方式，x86和x86_64定义了不同的函数调用规约(calling convention)。相比x86_64将参数传入通用寄存器的方式，x86将参数压入调用栈中，x86下对应foo函数传参的汇编指令，有以下形式的输

出：

```
sub $0x8, %esp
mov $0x5, -0x4(%ebp)
mov $0x2, -0x8(%ebp)
```

参数的调用栈位置通过ebp保存的栈帧栈底地址索引，栈从内存高地址向低地址生长，所以索引值为负数，减少esp寄存器的值表示扩展栈帧。

万事具备，是时候将执行控制权交给foo函数了，call指令完成交接任务：

```
0x000000000040052f <main+14>:     e8 d2 ff ff ff    callq  0x400506 <foo>
```

一条call指令，完成了两个任务：

1. 将调用函数(main)中的下一条指令(这里为0x400534)入栈，被调函数返回后将取这条指令继续执行，64位rsp寄存器的值减8
2. 修改指令指针寄存器rip的值，使其指向被调函数(foo)的执行位置，这里为0x400506

执行完start命令后，现在程序停在0x400522的位置，下面我们通过gdb的si指令，让程序执行完call指令：

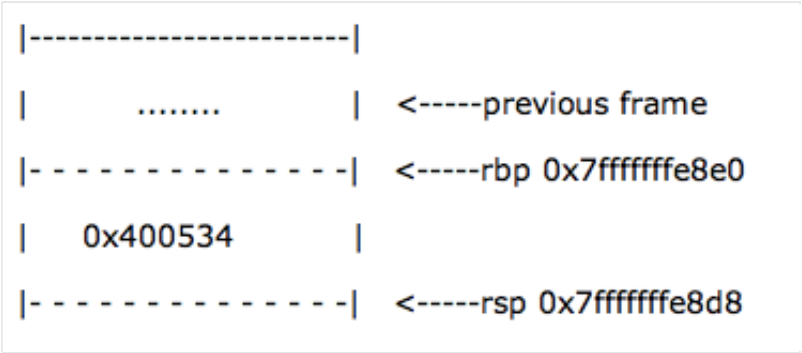
```
(gdb) si 3
foo (a=0, b=4195328) at func_call.c:8
8    {
```

```
(gdb)
```

此时我们再来看rsp、rbp寄存器的值，它们保存了程序实际用到的物理内存地址：

```
(gdb) info registers rbp rsp
rbp            0x7fffffffe8e0    0x7fffffffe8e0
rsp            0x7fffffffe8d8    0x7fffffffe8d8
(gdb)
```

main函数君的执行到此就暂时告一段落了，此时func_call的调用栈情况如下：



相关寄存器信息如下：

esi: 0x5 edi: 0x2

foo函数

foo函数被执行之后，我们使用disassemble命令显示其汇编指令：

```
(gdb) disassemble /rm
Dump of assembler code for function foo:
8    {
0x0000000000400506 <foo+0>:     55              push   %rbp
0x0000000000400507 <foo+1>:     48 89 e5        mov    %rsp,%rbp
0x000000000040050a <foo+4>:     48 83 ec 08     sub    $0x8,%rsp
0x000000000040050e <foo+8>:     89 7d fc        mov    %edi,-0x4(%rbp)
0x0000000000400511 <foo+11>:    89 75 f8        mov    %esi,-0x8(%rbp)

9        return bar(a, b);
0x0000000000400514 <foo+14>:     8b 75 f8       mov    -0x8(%rbp),%esi
0x0000000000400517 <foo+17>:     8b 7d fc       mov    -0x4(%rbp),%edi
0x000000000040051a <foo+20>:     e8 cd ff ff ff    callq  0x4004ec <bar>

10    }
0x000000000040051f <foo+25>:     c9    leaveq
0x0000000000400520 <foo+26>:     c3    retq

End of assembler dump.
```

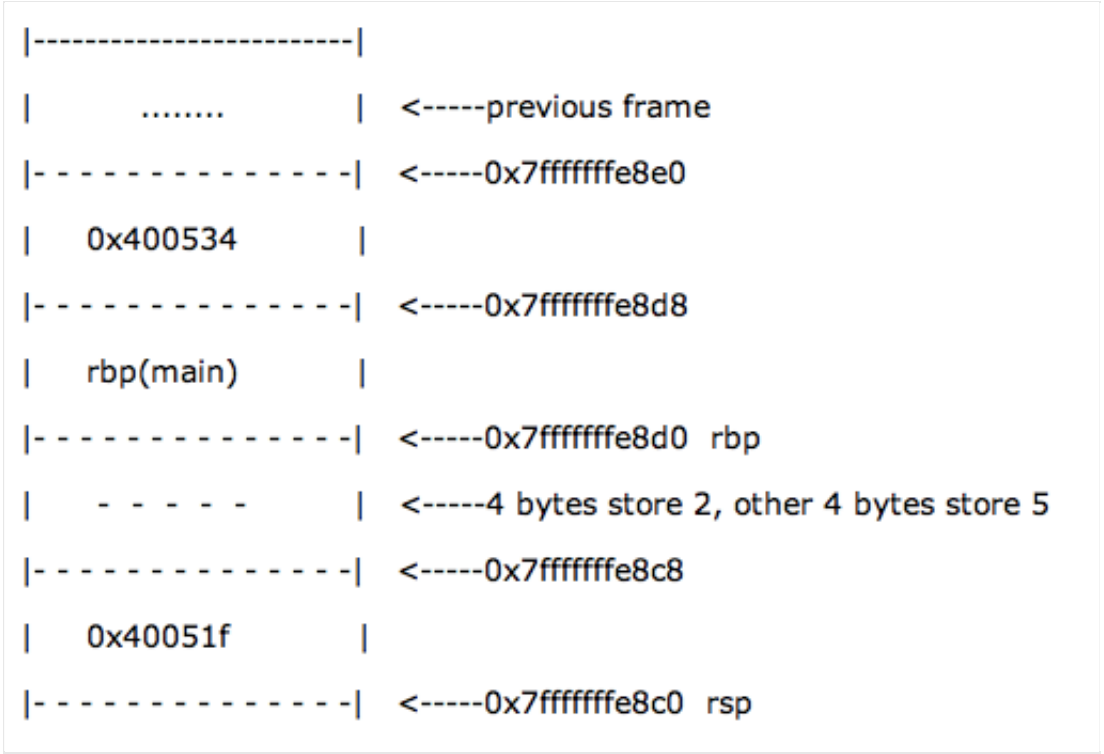
```
(gdb)
```

前面两条指令将main函数栈帧的栈底地址入栈，建立foo函数的栈帧。接着的三条指令扩展栈帧，将传入的参数存为函数内局部变量。最后三条指令与bar函数调用相对应，也是先将参数传入esi、edi寄存器，然后执行call指令。

继续执行si命令，让程序执行到call指令的位置：

```
(gdb) si 8
bar (c=32767, d=-139920736) at func_call.c:2
2    {
(gdb) info registers rbp rsp
rbp            0x7fffffffe8d0    0x7fffffffe8d0
rsp            0x7fffffffe8c0    0x7fffffffe8c0
(gdb)
```

foo函数调用bar函数之后，bar函数执行之前，调用栈信息如下：



相关寄存器信息如下：

esi: 0x5 edi: 0x2

bar函数

此时程序执行至bar函数，同样，我们先用disassemble看一下bar函数的汇编指令：

```
(gdb) disassemble /rm
Dump of assembler code for function bar:
```

```
2    {
0x00000000004004ec <bar+0>:     55          push   %rbp
0x00000000004004ed <bar+1>:     48 89 e5    mov    %rsp,%rbp
0x00000000004004f0 <bar+4>:     89 7d ec    mov    %edi,-0x14(%rbp)
0x00000000004004f3 <bar+7>:     89 75 e8    mov    %esi,-0x18(%rbp)

3        int e = c + d;
0x00000000004004f6 <bar+10>:     8b 55 e8    mov    -0x18(%rbp),%edx
0x00000000004004f9 <bar+13>:     8b 45 ec    mov    -0x14(%rbp),%eax
0x00000000004004fc <bar+16>:     01 d0       add    %edx,%eax
0x00000000004004fe <bar+18>:     89 45 fc    mov    %eax,-0x4(%rbp)

4        return e;
0x0000000000400501 <bar+21>:     8b 45 fc    mov    -0x4(%rbp),%eax

5    }
0x0000000000400504 <bar+24>:     c9    leaveq
0x0000000000400505 <bar+25>:     c3    retq

End of assembler dump.
(gdb)
```

对于最前面两条指令我们应该很熟悉了：将foo函数栈帧的栈底地址入栈，建立bar函数的栈帧。但后面两条指令与foo函数中对应位置的指令就不一样了，这里为什么不扩展栈帧，不像foo函数汇编指令那样将参数的值存入调用栈呢?

原因就是bar函数是最后一个被调用的函数了，foo函数中的局部变量在bar函数返回后还有可能被操作，而bar函数的局部变量已失去保存的必要。以上“{}”中剩余的指令利用edx和eax寄存器完成加法操作，最后结果保存在eax寄存器中，以作为结果返回。

至此，调用栈信息如下：


|----------|
| ..... | <-----previous frame
|----------| <-----0x7fffffffe8e0
| 0x400534 |
|----------| <-----0x7fffffffe8d8
| rbp(main) |
|----------| <-----0x7fffffffe8d0
| - - - - - | <-----first 4 bytes store 2, other 4 bytes store 5
|----------| <-----0x7fffffffe8c8
| 0x40051f |
|----------| <-----0x7fffffffe8c0
| rbp(foo) |
|----------| <-----0x7fffffffe8b8 rbp rsp
| - - - - - | <-----first 4 bytes store 7
|----------| <-----0x7fffffffe8b0
|
|----------| <-----0x7fffffffe8a8
| - - - - - | <-----first 4 bytes store 2, other 4 bytes store 5
|----------| <-----0x7fffffffe8a0

相关寄存器信息如下:

esi: 0x5 edi: 0x2 edx: 0x5 eax: 0x7

这时我们再来使用gdb的x命令查看内存信息：

```
(gdb) x/16x 0x7fffffffe8a0
0x7fffffffe8a0:    0x00000005    0x00000002    0x00400595    0x00000000
0x7fffffffe8b0:    0xf7ffa658    0x00000007    0xffffe8d0    0x00007fff
0x7fffffffe8c0:    0x0040051f    0x00000000    0x00000005    0x00000002
0x7fffffffe8d0:    0xffffe8e0    0x00007fff    0x00400534    0x00000000
(gdb)
```

以上命令显示16个4bytes内存地址指示的值，且值以十六进制显示。比较下，看这里的输出与上面的调用栈信息是否一致？

函数返回过程

函数调用过程对应着调用栈的建立，而函数返回则是进行调用栈的销毁，返回比调用过程简单多了，毕竟破坏比建设来的容易。在main、foo和bar函数的汇编显示中，我们都可以看到leave和ret两条指令：

```
0x0000000000400504 <bar+24>:    c9    leaveq
0x0000000000400505 <bar+25>:    c3    retq
```

leave指令等价于以下两条指令：

```
mov %rbp, %rsp
pop %rbp
```

这两条指令将bp和sp寄存器中的值还原为函数调用前的值，是函数开头两条指令的逆向过程。ret指令修改了ip寄存器的值，将其设置为原函数栈帧中将要执行的指令地址。bar函数的leave和ret执行完之后，调用栈信息变为：

rip寄存器的值为0x40051f

剩余的函数返回过程类似，直至所有函数执行完成、调用栈被销毁。

小结

本文通过一个简单的函数调用实例，结合gdb单步调试和反汇编工具，对函数调用的底层实现过程进行了分析。

修改sp、bp寄存器记录栈帧的高、低地址，以此完成函数调转；

push/mov操作保存caller变量、指令信息，保证callee返回之后caller继续正常执行；

……

栈这种简单的数据结构优雅地完成了支撑计算机程序执行的任务。

我们可以参照这样的思路，在编码实现功能需求时，分析所要实现的功能，选择恰当的数据结构和实现方式，力求做到优雅、简洁。

本文基于Suse11sp1(x86_64)，该发行版可从这里下载。

```
# cat /etc/SuSE-release;uname -r
SUSE Linux Enterprise Desktop 11 (x86_64)
VERSION = 11
PATCHLEVEL = 1
```

```
2.6.32.12-0.7-default
```

Reference: 函数调用

Chapter 5, the stack, Self-service Linux