深入理解函数调用过程

2016/1/18

renyl

1 介绍

1） 本文主要通过一个简单实例来引发对函数调用过程的说明。
2） 为了深入理解函数调用过程，主要从汇编代码的角度去剖析它。
3） 通过一个实例来扩展对函数调用的理解。

2 实例与分析

2.1 源码

1）源码说明：

调用两个简单的自定义函数。

2）源码显示：

test.c

```
#include <stdio.h>
int foo();
int bar();
int main(int argc,char *argv[])
{
        int result=foo(2,3,4);
        printf("result=%d\n",result);
        return 0;
}

int foo(int a,int b)
{
        return bar(a);
}

int bar(int c,int d)
{
        int e=c+d;
        return e;
}
```

注：

由于 Old Style C 语法的存在，并不是所有函数声明都包含函数原型。例如函数声明 int foo();没有明确指出参数类型和个数，所以不算函数原型，这个声明提供给编译器的信息只有函数名和返回值类型。如果在这样的声明之后调用该函数，编译器将不做参数类型检查和自动转换。

2.2 结果

编译后运行，结果如下：

```
[root@localhost assemble]# gcc -Wall -g test.c -o test
[root@localhost assemble]# ./test
result=5
```

2.3 疑问

1）gcc 编译时，参数类型不匹配，为什么没有任何警告？
2）执行结果为什么是 5，不是 6 或是 7？

2.3 分析

在进行分析之前，先来了解下进程地址空间的分布：

说明：
1） 栈空间是向低地址增长的，主要是用来保存函数栈帧。
2） 栈空间的大小很有限，仅有区区几 MB 大小。

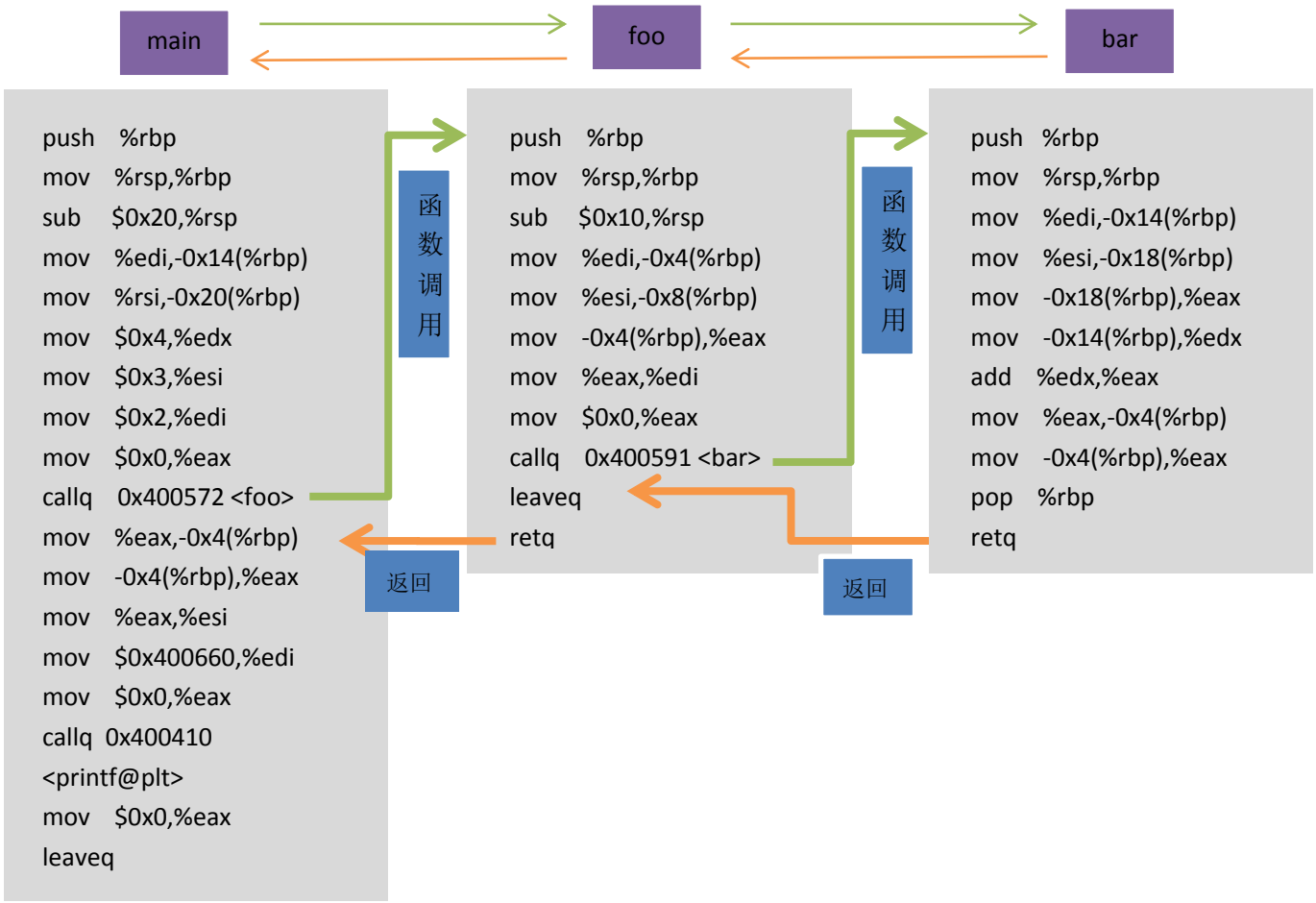
接下来，首先使用 gdb（或者 objdump -dS test）对上述代码进行反汇编，如下所示：

```
[root@localhost assemble]#gdb  test
GNU gdb (GDB) Red Hat Enterprise Linux (7.5.1-34.el7)
…
(gdb) start
Temporary breakpoint 1 at 0x40053b: file test.c, line 9.
```

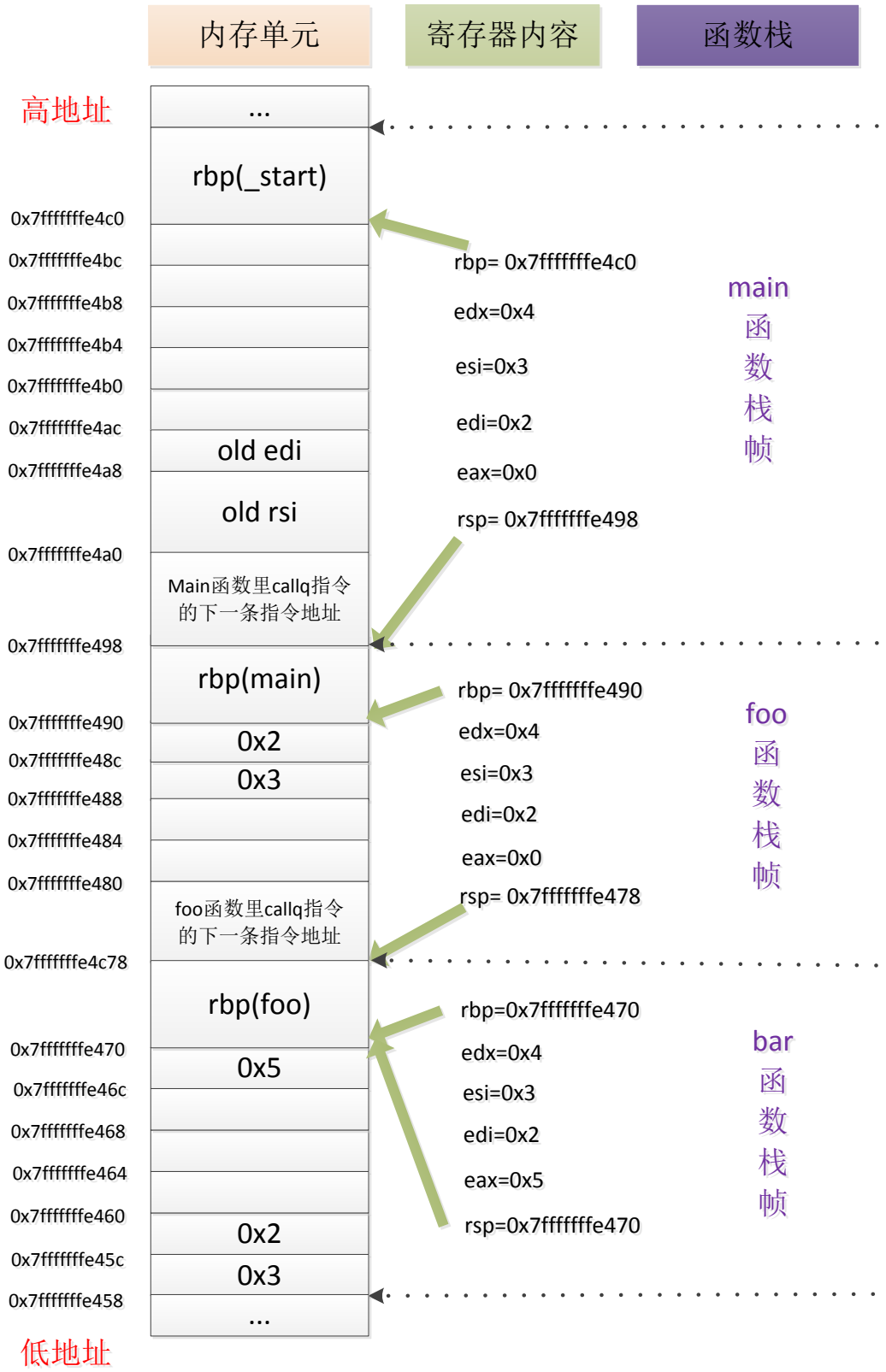
```
Starting program: /home/renyl/testdir/assemble/test

Temporary breakpoint 1, main (argc=1, argv=0x7fffffffe5d8) at test.c:9
9                       int result=foo(2,3,4);
Missing separate debuginfos, use: debuginfo-install glibc-2.16-29.el7.x86_64
(gdb) disassemble main
Dump of assembler code for function main:
   0x000000000040052c <+0>:     push   %rbp
   0x000000000040052d <+1>:     mov    %rsp,%rbp
   0x0000000000400530 <+4>:     sub    $0x20,%rsp
   0x0000000000400534 <+8>:     mov    %edi,-0x14(%rbp)
   0x0000000000400537 <+11>:    mov    %rsi,-0x20(%rbp)
=> 0x000000000040053b <+15>:    mov    $0x4,%edx
   0x0000000000400540 <+20>:    mov    $0x3,%esi
   0x0000000000400545 <+25>:    mov    $0x2,%edi
   0x000000000040054a <+30>:    mov    $0x0,%eax
   0x000000000040054f <+35>:    callq    0x400572 <foo>
   0x0000000000400554 <+40>:    mov    %eax,-0x4(%rbp)
   0x0000000000400557 <+43>:    mov    -0x4(%rbp),%eax
   0x000000000040055a <+46>:    mov    %eax,%esi
   0x000000000040055c <+48>:    mov    $0x400660,%edi
   0x0000000000400561 <+53>:    mov    $0x0,%eax
   0x0000000000400566 <+58>:    callq    0x400410 <printf@plt>
   0x000000000040056b <+63>:    mov    $0x0,%eax
   0x0000000000400570 <+68>:    leaveq
   0x0000000000400571 <+69>:    retq
End of assembler dump.
(gdb) disassemble  foo
Dump of assembler code for function foo:
   0x0000000000400572 <+0>:     push   %rbp
   0x0000000000400573 <+1>:     mov    %rsp,%rbp
   0x0000000000400576 <+4>:     sub    $0x10,%rsp
   0x000000000040057a <+8>:     mov    %edi,-0x4(%rbp)
   0x000000000040057d <+11>:    mov    %esi,-0x8(%rbp)
   0x0000000000400580 <+14>:    mov    -0x4(%rbp),%eax
   0x0000000000400583 <+17>:    mov    %eax,%edi
   0x0000000000400585 <+19>:    mov    $0x0,%eax
   0x000000000040058a <+24>:    callq  0x400591 <bar>
   0x000000000040058f <+29>:    leaveq
   0x0000000000400590 <+30>:    retq
End of assembler dump.
(gdb) disassemble  bar
Dump of assembler code for function bar:
   0x0000000000400591 <+0>:     push   %rbp
   0x0000000000400592 <+1>:     mov    %rsp,%rbp
   0x0000000000400595 <+4>:     mov    %edi,-0x14(%rbp)
   0x0000000000400598 <+7>:     mov    %esi,-0x18(%rbp)
   0x000000000040059b <+10>:    mov    -0x18(%rbp),%eax
   0x000000000040059e <+13>:    mov    -0x14(%rbp),%edx
   0x00000000004005a1 <+16>:    add    %edx,%eax
   0x00000000004005a3 <+18>:    mov    %eax,-0x4(%rbp)
```

```
   0x00000000004005a6 <+21>:    mov    -0x4(%rbp),%eax
   0x00000000004005a9 <+24>:    pop    %rbp
   0x00000000004005aa <+25>:    retq
End of assembler dump.
```

把上述用 gdb 生成的汇编代码按照程序运行走向作成流程图，如下所示：



通过对上述汇编代码进行分析以及使用工具 gdb 进行调试，制作各个函数栈帧在内存中的分布情况以及所使用到的寄存器及其存储值，如下所示：



注：

准确来说 main 函数并不是程序的入口点，_start 函数才是程序的入口点（_start 函数由目标文件 /usr/lib/crt1.o 提供，由 gcc 在链接阶段调用），main 函数是被_start 函数调用的。

在进一步分析之前，需要了解下内存寻址方式以及某些特殊寄存器的用途。

1） 内存寻址

访问内存时在指令中可以用多种方式表示内存地址。

通用格式为：

ADDRESS_OF_OFFSET(%BASE_OF_OFFSET,%INDEX,MULTIPLIER)

计算方法为：

FINAL_ADDRESS= ADDRESS_OF_OFFSET+BASE_OF_OFFSET+(INDEX * MULTIPLIER)

其中，ADDRESS_OF_OFFSET 和 MULTIPLIER 必须是常数，BASE_OF_OFFSET 和 INDEX 必须是寄存器。在有些寻址方式中会省略这 4 项中的某些项，相当于这些项是 0。

访问内存的寻址方式如下：

a） 直接寻址（Direct Access Address）： 只使用 ADDRESS_OF_OFFSET 寻址。mov ADDRESS %eax 表示把 ADDRESS 地址处的 32 位数传送给寄存器 eax。

b） 变址寻址（Indexed Acces Address）：用于访问数组元素比较方便。假设 array_address 表示数组首地址，寄存器 ebx 保存着 array_address 数组下标，那么 mov array_address(,%ebx,4), %eax 表示把 ebx 保存的下标数组元素传送给寄存器 eax。

c） 间接寻址（Indirect Acces Address）：只使用 BASE_OF_OFFSET 寻址。mov (%eax), %ebx 表示把 eax 寄存器保存的地址处的 32 位数传送给寄存器 ebx。

d） 基址寻址（Base Pointer Access Address）：只使用 ADDRESS_OF_OFFSET 和 BASE_OF_OFFSET，用于访问结构体成员比较方便。假设一个结构体的基地址保存在寄存器 eax 中，其中一个成员变量在结构体内的偏移量是 4 字节，那么 mov 4(%eax), %ebx 表示把这个成员变量放入寄存器 ebx 中。

e） 立即数寻址（Immediate Access）：就是指令中有一个操作数是立即数，如 mov $1 %eax 表示把整数 1 传递给寄存器 eax。

f） 寄存器寻址（Register Access Address）：就是指令中有一个操作数是寄存器，如 mov $2,%ebx 和 mov (%eax), %ebx 都算作寄存器寻址。

2） 特殊寄存器的说明

a） rsp 寄存器为栈顶寄存器，始终指向函数栈的栈顶。

b） rbp 寄存器为栈底寄存器，保存着当前函数栈的栈底。

c） eip 寄存器为程序计数器，保存着下一条执行指令的地址。

注：寄存器 ax、bx 等为 16 位寄存器， 寄存器 eax、ebx 等（以 e 开头）为 32 位寄存器，寄存器 rax、rbx 等（以 r 开头）为 64 位寄存器。

接下来就通过对 main 函数的汇编代码分析来理解 main 函数栈帧：

1） push %rbp

说明：

把寄存器 rbp 压栈。此时 rbp 寄存器保存的是调用者（_start 函数）栈帧的栈底，main 函数返回后，_start 函数需要用到寄存器 rbp 中保存的值，因此需要压栈。

2） mov %rsp,%rbp

说明：（寄存器寻址）

把寄存器 rsp 中的值赋值给寄存器 rbp。现在寄存器 rbp 保存的是 main 函数栈帧的栈底

3）sub　　$0x20,%rsp

说明：（立即数寻址）

把寄存器 rsp 中的值减去 32 个字节。现在寄存器 rsp 指向低于当前 32 字节的地址处

4）mov　　%edi,-0x14(%rbp)

说明：（基址寻址）

把寄存器 edi 中的值保存在栈帧中。因为寄存器 edi 接下来要被使用，为了在 main 函数返回后，寄存器 edi 能够恢复到原来的值，需要保存在栈帧中。

5）mov　　%rsi,-0x20(%rbp)

说明：（基址寻址）

把寄存器值 rsi 保存在栈帧中。因为寄存器 rsi 接下来要被使用，为了在 main 函数返回后，寄存器 rsi 能够恢复到原来的值，需要保存在栈帧中。

6）mov　　$0x4,%edx

说明：（立即数寻址）

把整数 4 放入寄存器 edx 中

7）mov　　$0x3,%esi

说明：（立即数寻址）

把整数 4 放入寄存器 esi 中

8）mov　　$0x2,%edi

说明：（立即数寻址）

把整数 2 放入寄存器 edi 中

9）mov　　$0x0,%eax

说明：（立即数寻址）

把整数 0 放入寄存器 eax 中

10）callq　　0x400572 <foo>

说明：（函数调用过程的关键）

调用 foo 函数。foo 函数调用完之后要返回到 callq 的下一条指令继续执行，因此 callq 指令会做两件事：

a）把 callq 指令的下一条指令地址 0x0000000000400554 压栈，同时寄存器 rsp 的值将减 8。

b）修改程序计数器 eip，使其指向 foo 函数的首地址，然后跳转到 foo 函数的开头执行。

注：foo 与 bar 函数的汇编代码跟 main 函数类似，就不分析了，结合上图的函数调用栈自行推断。

接下结合 foo 函数重点对函数调用的返回指令 leaveq 和 retq 进行分析：

1） leaveq 指令：

说明：

foo 函数的开头有两条指令（push　%rbp；mov　%rsp，%rbp），leaveq 就是这两条指

令的逆操作。分为两步：

a） mov %rbp,%rsp ：

把寄存器 rbp 的值赋给寄存器 rsp，让寄存器 rsp 指向保存 main 函数栈底的地址

b） pop %rbp

把寄存器 rsp 所指向的内存单元值赋值给 rbp，这样 rbp 现在就指向 main 函数的栈底。同时寄存器 rsp 加 8，此时 rsp 指向调用函数 main 的返回地址。

2） retq 指令：

说明：

main 函数调用 foo 时需要 callq 指令，foo 函数返回时就需要 retq 指令，它是 callq 指令的逆操作。同样需要分为两步：

a） 把 rsp 指向调用函数的返回地址赋值给程序计数器 eip，同时 rsp 寄存器加 8。

b） 程序返回到 eip 寄存器所指向的地址继续执行。

2.4 总结

通过对函数调用的汇编代码进行分析，得出如下：

1） 参数可以通过寄存器直接传递，不需要通过压栈传递（当参数变量数量较多时，寄存器无法保存所有变量，这个时候需要通过压栈传递），参数压栈时从右向左依次压栈，而被调用函数的参数是从栈帧的低地址向高地址去取，因此上述代码执行结果是 5。

2） 在每个函数的栈帧中，寄存器 rbp 指向栈底，寄存器 rsp 指向栈顶，在函数执行过程中 rsp 随着压栈和出栈操作会发生变化，而 rbp 却是不变的。

3） 函数返回值是通过 eax 寄存器传递的。

上述的这些规则并不是体系结构所强加的，寄存器 rbp 并不是必须这么用，函数的参数和返回值也不是必须这么传，只是操作系统和编译器选择了以这样的方式实现 C 代码中的函数调用，这称为“调用约定”（Calling Convention）。

接下来，通过一个实例来说明对函数调用理解的应用。

3 应用

3.1 源码

1） 源码说明：

a） comp.c ：如果变量 a 与 b 相等，向终端打印“OK”，否则打印“Soryy”。

b） preload.c：仿照 write 系统调用重写一个 write 函数

2） 源码显示：

comp.c

```
#include <stdio.h>
#include <unistd.h>
int main(int argc,char *argv[])
{
    int a=1,b=2;
    if ( a != b )
    {
        write(1,"Sorry\n",sizeof("Sorry\n"));
        return 0;
    }
        write(1,"OK\n",sizeof("OK\n"));
        return 1;
}
```

preload.c

```
#define _GNU_SOURCE
#include <stdarg.h>
#include <dlfcn.h>
#include <unistd.h>
#include <stdio.h>

static ssize_t (*_write)(int,const void*,size_t)=NULL;

ssize_t write(int fd,const void* buf,size_t count)
{
        if(_write==NULL)
        {
           //获取 libc.so 库中 write 系统调用的地址
            _write=(ssize_t(*)(int,const
void*,size_t))dlsym(RTLD_NEXT,"write");
            if(_write==NULL)
                return 1;

          //修改函数的返回地址
        int b=7;
        __asm__ __volatile__(
```

```
                        "movq 0x8(%%rbp),%%rax\n"
                        "addq %%rcx, %%rax\n"
                        "movq %%rax, 0x8(%%rbp)\n"
                        :
                        :"c"(b)
                        :"memory"
                        );
        return 2;
        }

        //调用 libc.so 库中的 write 系统调用
        (*_write)(fd,buf,count);
        return 3;
}
```

3.2 结果

编译后运行，结果如下：

```
[root@localhost preload]# gcc -g comp.c -o comp
[root@localhost preload]# gcc -shared -fPIC preload.c  -o preload.so  -ldl
[root@localhost preload]# LD_PRELOAD=./preload.so  ./comp
OK
```

说明：

1） 把 preload.c 编译成一个动态共享库，以便别的函数调用。

2） LD_PRELOAD 是 Linux 下的一个环境变量，它允许你定义程序运行前优先加载的动态链接库。这个功能主要就是用来有选择性的载入不同动态链接库中的相同函数。

3） 由于设置 LD_PRELOAD 环境变量，程序 comp 在运行是会优先调用已编译好的 preload.so 动态库中的 write 函数。

3.3 疑问

打印到终端的不是“Sorry”，居然是“OK”，如何做到的呢？

3.4 分析

首先使用命令 objdump -dS comp 进行反汇编，显示如下：

```
…
…
00000000004004c4 <main>:
#include <stdio.h>
#include <unistd.h>
```

```
int main(int argc,char *argv[])
{
  4004c4:       55                      push   %rbp
  4004c5:       48 89 e5                 mov    %rsp,%rbp
  4004c8:       48 83 ec 20               sub    $0x20,%rsp
  4004cc:       89 7d ec                  mov    %edi,-0x14(%rbp)
  4004cf:       48 89 75 e0               mov    %rsi,-0x20(%rbp)
        int a=1,b=2;
  4004d3:       c7 45 f8 01 00 00 00        movl   $0x1,-0x8(%rbp)
  4004da:       c7 45 fc 02 00 00 00        movl   $0x2,-0x4(%rbp)

        if ( a != b )
  4004e1:       8b 45 f8                  mov    -0x8(%rbp),%eax
  4004e4:       3b 45 fc                  cmp    -0x4(%rbp),%eax
  4004e7:       74 1b                    je     400504 <main+0x40>
        {
                write(1,"Sorry\n",sizeof("Sorry\n"));
  4004e9:       ba 07 00 00 00          mov    $0x7,%edx
  4004ee:       be 18 06 40 00          mov    $0x400618,%esi
  4004f3:       bf 01 00 00 00           mov    $0x1,%edi
  4004f8:       e8 cb fe ff ff             callq  4003c8 <write@plt>//第一个
write 调用
                return 0;
  4004fd:       b8 00 00 00 00           mov    $0x0,%eax
  400502:       eb 19                  jmp    40051d <main+0x59>
        }

                write(1,"OK\n",sizeof("OK\n"));
  400504:       ba 04 00 00 00          mov    $0x4,%edx
  400509:       be 1f 06 40 00           mov    $0x40061f,%esi
  40050e:       bf 01 00 00 00           mov    $0x1,%edi
  400513:       e8 b0 fe ff ff             callq  4003c8 <write@plt> //第二个
write 调用
                return 1;
  400518:       b8 01 00 00 00           mov    $0x1,%eax
}
  40051d:       c9                     leaveq
  40051e:       c3                     retq
  40051f:       90                      nop
…
…
```

说明：

1）通过汇编代码可以看出第一个 wirte 函数调用返回后会执行这条指令（mov $0x0,%eax），然后接着执行指令（jmp 40051d <main+0x59>），最后程序退出。这样第二个 write 函数就不会被调用了，“OK”也就不会被打印到终端了。

2）要想让代码能够执行到第二个 write 函数，就需要使用环境变量 LD_PRELOAD 设置优先加载的动态库，让程序执行 write 函数时调用我们编写的 write 函数。

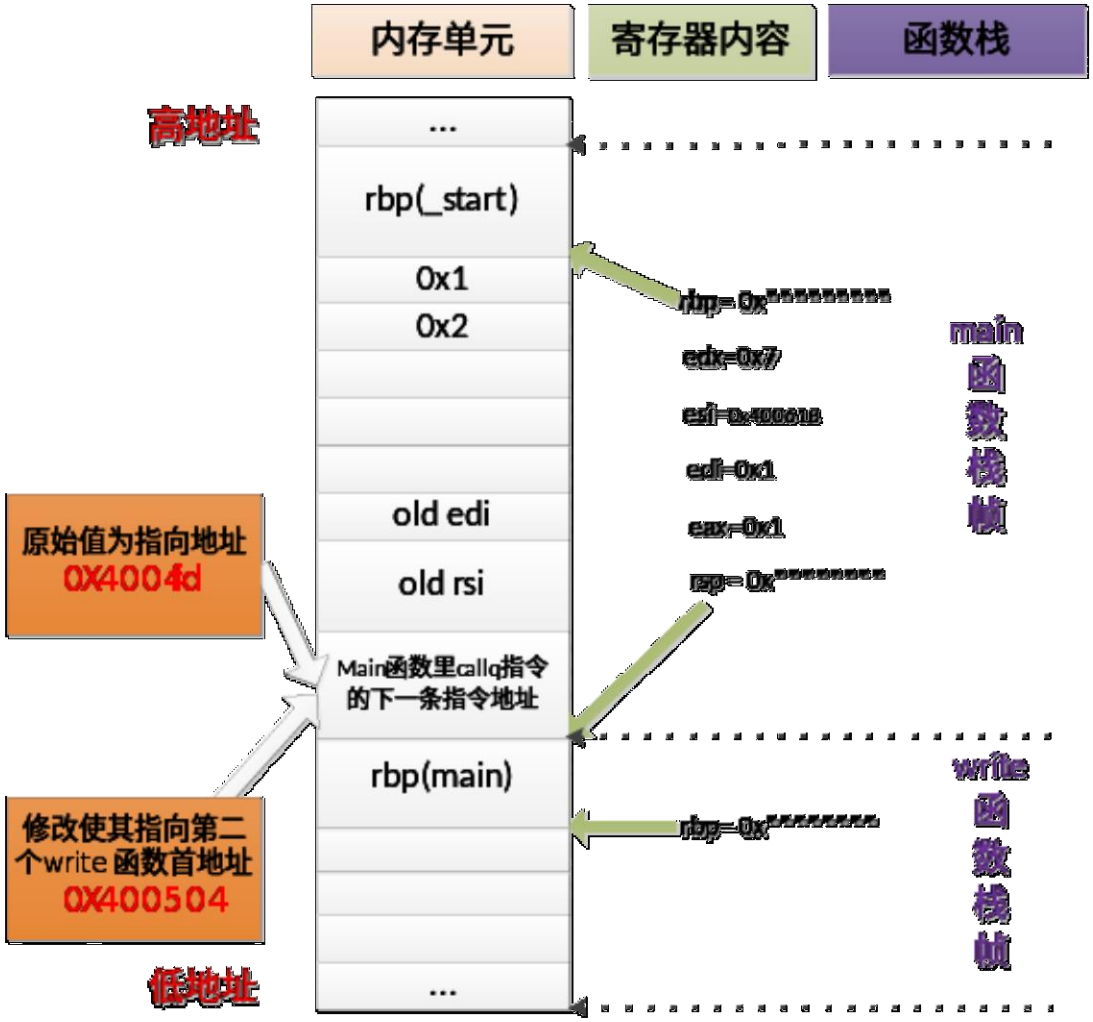
3） 要想在终端打印“OK”，我们编写的 write 函数逻辑处理应该如下：

a） 当程序第一次调用 write 函数时，其返回地址要指向指令（mov $0x4,%edx）的地址 0X400504（第二个 write 函数调用的首地址），而不是指向指令（mov $0x0,%eax）的地址 0X4004fd。

b） 当程序第二次调用 write 函数时，让其执行 libc.so 库中的 write 系统调用，这样将正常的执行并向终端打印出“OK”。

现在思路已经有了，重点是如何让程序第一次调用 write 函数后返回的地址指向第二个 write 函数调用的首地址 0X400504，而不是指向 0X4004fd。

仅根据上述汇编代码即可制作这段代码的函数调用栈帧，如下所示：



说明：

1） 从上图可知，在 write 函数返回前，需要修改函数返回后的下一条执行指令地址。

2） 在 write 函数里可通过内联汇编来修改下一条执行指令地址，该地址永远都存储在 rbp（main）的上方内存单元中，因此用 0x8(%rbp)即可获得到该地址。

注：32 位系统的话用 0x4(%rbp)获取这个地址。

3.5 总结

1） 通过对这个实例的分析，进一步加深了对函数调用过程的理解。

2） 通过内联汇编代码可以修改函数返回后下一条执行指令的地址，但是在什么地方修改以及如何获取原始返回地址需要对函数调用过程有着深入理解。