**经典算法--约瑟夫环问题的三种解法**

2018-05-17 17:03:07 [西邮陈冠希](https://me.csdn.net/weixin_38214171) 阅读数 43672更多

分类专栏： [数据结构与算法](https://blog.csdn.net/weixin_38214171/article/category/7524101)

版权声明：本文为博主原创文章，遵循[CC 4.0 BY-SA](http://creativecommons.org/licenses/by-sa/4.0/)版权协议，转载请附上原文出处链接和本声明。

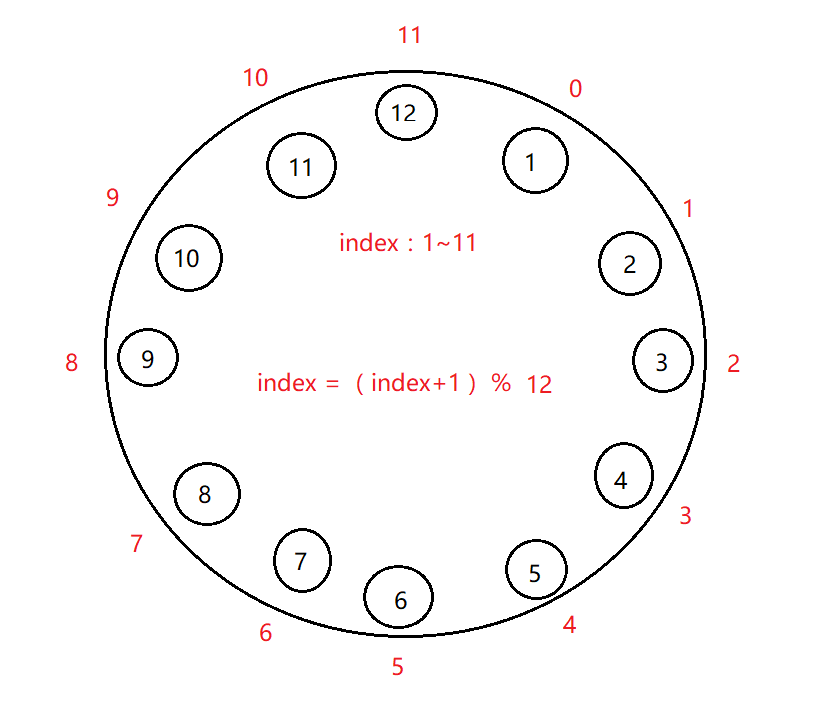
本文链接：<https://blog.csdn.net/weixin_38214171/article/details/80352921>

约瑟夫环问题，这是一个很经典算法，处理的关键是：伪链表

问题描述：N个人围成一圈，从第一个人开始报数，报到m的人出圈，剩下的人继续从1开始报数，报到m的人出圈；如此往复，直到所有人出圈。（模拟此过程，输出出圈的人的序号）

在数据结构与算法书上，这个是用链表解决的。我感觉链表使用起来很麻烦，并且这个用链表处理起来也不是最佳的。

我画了一个图用来理解：



有如下问题需要首先考虑：

1、“圈”怎样形成？

    以上图为例：下标从0开始，当下标为11的时候，再加1，就应该回到0。

index = （index+1） % count；

2、怎样处理？数组or链表or其他方法？

    链表使用起来很笨重，我们用循环数组的方法。

**解法一程序分析：**

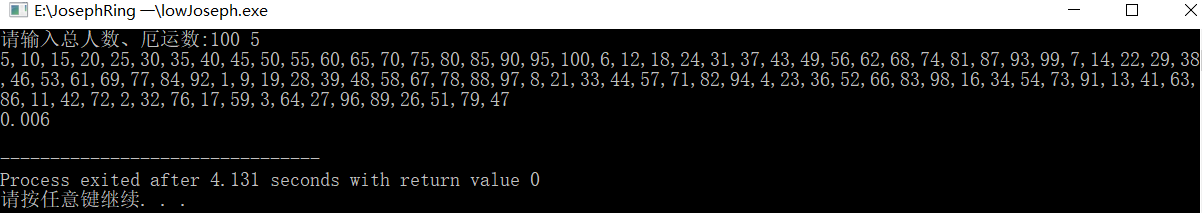
循环的开始和结束：循环的结束取决于圈内是否还有“人”，可以用一个变量alive表示初始人数，每一次出圈，alive - 1。判断alive是否非0即可。

    while(alive > 0)

每一次循环，就是“过”一个人，但是，这个人有两种不同的状态：在圈内和不在圈内；在圈内就报数，number+1，不在圈内就不参与报数，number不变。

假设有N个int元素的数组，每一个int元素表示一个“人”；并且，取值为0和1, 1表示在圈内，0表示不在圈内，所以，如果这个人在圈内，number + 1；如果这个人不在圈内，number + 0。那么，在报数的时候，不需要考虑这个人在不在圈内（每一个人都需要加1或加0，所以，可以在这块优化一下程序）。

1. void joseph(int count, int doom) {
2. int alive = count; //幸存人数
3. int number = 0; //计数，当number==doom时，淘汰这个人
4. int index = 0; //下标，为总人数-1
5. int \*circle = NULL; //根据需求设为循环数组，存储每个人
7. //用calloc()函数申请得到的空间，自动初始化每个元素为0
8. //所以，0表示在这个人在约瑟夫环内，1表示这个人出圈，即“淘汰”
9. circle = (int \*) calloc(sizeof(int), count);
11. //只要幸存人数大于0，则一直进行循环
12. while(alive > 0) {
13. number += 1- circle[index]; //每轮到一个人报数，不管是"0"还是"1"都进行计数
14. if(number == doom) { //当number==doom时，就要淘汰当前这个人
15. /\*
16. 淘汰一个人需要做四步操作：
17. 1、输出这个人的位置
18. 2、把这个人的状态从在圈内"0"改为不在圈内"1"
19. 3、幸存人数alive--
20. 4、 计数器number归零
21. \*/
22. alive == 1 ? printf("%d", index+1) : printf("%d,", index+1);
23. circle[index] = 1;
24. alive--;
25. number = 0;
26. }
27. //与总人数count取余，则可以使index在0~count-1之间 一直循环，达到循环数组的目的
28. index = (index +1) % count;
29. }
30. printf("\n");
32. free(circle); //结束后一定要释放circle所申请的空间
33. }



**解法二程序分析：**

解法二在解法一的基础上进行了优化，对出圈的人的节点进行删除，可以减少时间复杂度。

假设，要删除的节点下标为curIndex，其前驱节点下标为preIndex；

circle[preIndex] = circle[curIndex];

    假设要删除的节点下标curIndex=3，那么circle[2] = circle[3] ，circle[2]  = 4，circle[2]之前等于3，现在等于4。即下标为3的第四个人已经被删除了。

怎样遍历？

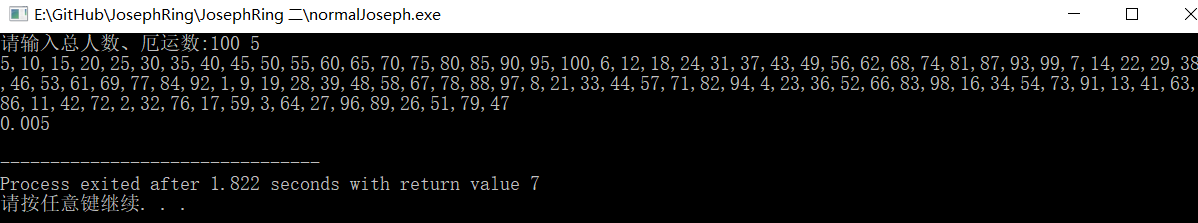
1. preIndex = curIndex;
3. curIndex = circle[curIndex];

但是，在出圈的时候，curIndex 和preIndex 的变化有别于上面的操作：

出圈时，curIndex 需要后移，preIndex 应该不动！

每次循环，直接报数，因为被删除的人（例如上面的第四个人）不可能进行报数了。

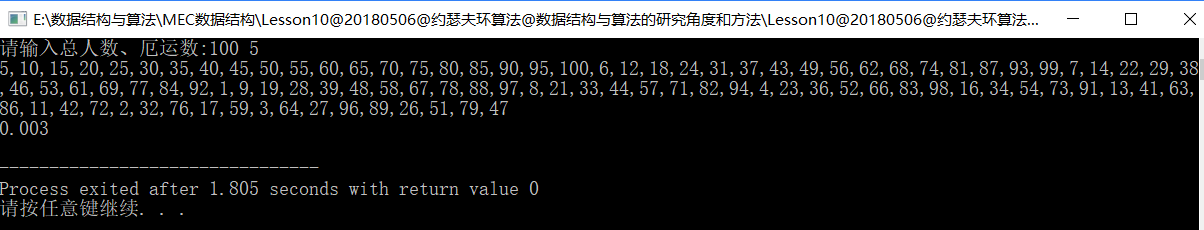
1. void joseph(int count, int doom) {
2. int alive = count; // 幸存人数
3. int number = 0; // 报数的数
4. int curIndex = 0; // 当前人下标
5. int preIndex = count - 1; // 前一个人下标
6. int \*circle = NULL;
7. int index;
9. circle = (int \*) malloc (sizeof(int) \* count);
10. //对circle数组进行初始化
11. for(index = 0; index < count; index++) {
12. circle[index] = (index + 1) % count;
13. }
15. while(alive > 0) {
16. number++;
17. if(number == doom) {
18. alive == 1 ? printf("%d", curIndex+1) : printf("%d,", curIndex+1);
19. alive--;
20. number = 0;
21. circle[preIndex] = circle[curIndex]; //出圈操作
22. } else {
23. preIndex = curIndex; //处理下一个人
24. }
25. curIndex = circle[curIndex];
26. }
28. free(circle);
29. }



**解法三程序分析：**

解法三里没有进行number报数，而是直接计算出需要移动的人数，然后定位到要出圈的人。

1. num = doom % alive - 1;
3. for(index = 0; index < (num == -1 ? alive - 1 : num); index++) {
4. preIndex = curIndex;
5. curIndex = circle[curIndex];
6. }
7. void joseph(int count, int doom) {
8. int alive = count; // 幸存人数
9. int curIndex = 0; // 当前人下标
10. int preIndex = count - 1; // 前一个人下标
11. int \*circle = NULL;
12. int index;
14. circle = (int \*) malloc(sizeof(int) \* count);
15. for(index = 0; index < count; index++) {
16. circle[index] = (index + 1) % count; // 初始化链表
17. }
19. while(alive > 0) { // 只要还有幸存者，就继续“杀”
20. int num = doom % alive - 1; // 直接计算出需要移动的人数，
21. // 直接定位到要出圈的人
22. for(index = 0; index < (num == -1 ? alive - 1 : num); index++) {
23. preIndex = curIndex;
24. curIndex = circle[curIndex];
25. }
26. // 该人出圈！
27. alive == 1 ? printf("%d", curIndex+1) : printf("%d,", curIndex+1);
28. alive--;
29. circle[preIndex] = circle[curIndex]; // 真正的出圈操作！
30. curIndex = circle[curIndex]; // 继续处理下一个人
31. }
32. // 这个算法比normalJoseph.c效率提高30%！
34. free(circle);
35. }



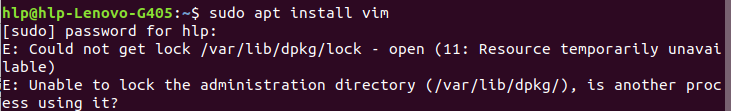
GitHub源码地址：

<https://github.com/yangchaoy259189888/JosephRing/>

[关于Ubuntu中Could not get lock /var/lib/dpkg/lock解决方案](https://www.cnblogs.com/yun6853992/p/9343816.html)

**1：我遇到的问题：**

在开机运行apt install vim 命令的时候，如下报错：



**2：参考博客：**

在Ubuntu中，有时候运用sudo  apt-get install 安装软件时，会出现一下的情况

E: Could not get lock /var/lib/dpkg/lock - open (11: Resource temporarily unavailable)  
E: Unable to lock the administration directory (/var/lib/dpkg/), is another process using it?

**3：问题原因：**主要是因为apt还在运行。

**4：解决方案：**杀死所有的apt进程。

　　1：查找所有apt相关的进程，并用命令杀死。

[复制代码](javascript:void(0);)

hlp@hlp-Lenovo-G405:~$ ps afx|grep apt

3284 pts/0 S+ 0:00 \\_ grep --color=auto apt

2869 ? Ss 0:00 /bin/sh /usr/lib/apt/apt.systemd.daily install

2873 ? S 0:00 \\_ /bin/sh /usr/lib/apt/apt.systemd.daily lock\_is\_held install

hlp@hlp-Lenovo-G405:~$ sudo kill -9 2873

hlp@hlp-Lenovo-G405:~$ sudo kill -9 2869

[复制代码](javascript:void(0);)

　　2：删除锁定文件

　　　　 锁定的文件会阻止 Linux 系统中某些文件或者数据的访问，这个概念也存在于 Windows 或者其他的操作系统中。

　　　　一旦你运行了 apt-get 或者 apt 命令，锁定文件将会创建于 /var/lib/apt/lists/、/var/lib/dpkg/、/var/cache/apt/archives/ 中。

　　　　这有助于运行中的 apt-get 或者 apt 进程能够避免被其它需要使用相同文件的用户或者系统进程所打断。当该进程执行完毕后，锁定文件将会删除。

　　　　所以：

　　　　　　1：移除对应目录下的锁文件：

　　　　　　2：强制重新配置软件包：

　　　　　　3：更新软件包源文件：

|  |  |
| --- | --- |
| 1  2  3 | hlp@hlp-Lenovo-G405:~$ sudo rm /var/lib/dpkg/lock  hlp@hlp-Lenovo-G405:~$ sudo dpkg --configure -a  hlp@hlp-Lenovo-G405:~$ sudo apt update |

　　3：问题解决，执行apt install 命令成功。

[深入理解C语言的函数调用过程](http://blog.chinaunix.net/uid-23069658-id-3981406.html)

    本文主要从进程栈空间的层面复习一下C语言中函数调用的具体过程，以加深对一些基础知识的理解。  
    先看一个最简单的程序：

点击(此处)折叠或打开

1. /\*test.c\*/
2. #include <stdio.h>
3. int foo1(int m,int n,int p)
4. {
5. int x = m + n + p;
6. return x;
7. }
8. int main(int argc,char\*\* argv)
9. {
10. int x,y,z,result;
11. x=11;
12. y=22;
13. z=33;
14. result = foo1(x,y,z);
15. printf("result=%d\n",result);
16. return 0;
17. }

    主函数main里定义了4个局部变量，然后调用同文件里的foo1()函数。4个局部变量毫无疑问都在进程的栈空间上，当进程运行起来后我们逐步了解一下main函数里是如何基于栈实现了对foo1()的调用过程，而foo1()又是怎么返回到main函数里的。为了便于观察的粒度更细致一些，我们对test.c生成的汇编代码进行调试。如下：

点击(此处)折叠或打开

1. .file "test.c"
2. .text
3. .globl foo1
4. .type foo1, @function
5. foo1:
6. pushl %ebp
7. movl %esp, %ebp
8. subl $16, %esp
9. movl 12(%ebp), %eax
10. movl 8(%ebp), %edx
11. leal (%edx,%eax), %eax
12. addl 16(%ebp), %eax
13. movl %eax, -4(%ebp)
14. movl -4(%ebp), %eax
15. leave
16. ret
17. .size foo1, .-foo1
18. .section .rodata
19. .LC0:
20. .string "result=%d\n"
21. .text
22. .globl main
23. .type main, @function
24. main:
25. pushl %ebp
26. movl %esp, %ebp
27. andl $-16, %esp
28. subl $32, %esp
29. movl $11, 16(%esp)
30. movl $22, 20(%esp)
31. movl $33, 24(%esp)
32. movl 24(%esp), %eax
33. movl %eax, 8(%esp)
34. movl 20(%esp), %eax
35. movl %eax, 4(%esp)
36. movl 16(%esp), %eax
37. movl %eax, (%esp)
38. call foo1
39. movl %eax, 28(%esp)
40. movl $.LC0, %eax
41. movl 28(%esp), %edx
42. movl %edx, 4(%esp)
43. movl %eax, (%esp)
44. call printf
45. movl $0, %eax
46. leave
47. ret
48. .size main, .-main
49. .ident "GCC: (GNU) 4.4.4 20100726 (Red Hat 4.4.4-13)"
50. .section .note.GNU-stack,"",@progbits

    上面的汇编源代码和最终生成的可执行程序主体结构上已经非常类似了：

|  |
| --- |
| [root@maple 1]# gcc -g -o test test.s  [root@maple 1]# objdump -D test >testbin  [root@maple 1]# vi testbin   //… 省略部分不相关代码  80483c0:       ff d0                               call   \*%eax   80483c2:      c9                                   leave   80483c3:      c3                                   ret    080483c4 :   80483c4:       55                                  push  %ebp   80483c5:      89 e5                               mov   %esp,%ebp   80483c7:      83 ec 10                          sub    $0x10,%esp   80483ca:      8b 45 0c                          mov    0xc(%ebp),%eax   80483cd:      8b 55 08                         mov   0x8(%ebp),%edx   80483d0:      8d 04 02                         lea    (%edx,%eax,1),%eax   80483d3:      03 45 10                         add    0x10(%ebp),%eax   80483d6:      89 45 fc                          mov    %eax,-0x4(%ebp)   80483d9:      8b 45 fc                          mov    -0x4(%ebp),%eax   80483dc:      c9                                   leave   80483dd:      c3                                   ret    080483de  :   80483de:      55                                     push   %ebp   80483df:      89 e5                                 mov   %esp,%ebp   80483e1:      83 e4 f0                            and    $0xfffffff0,%esp   80483e4:      83 ec 20                           sub    $0x20,%esp   80483e7:      c7 44 24 10 0b 00 00       movl   $0xb,0x10(%esp)   80483ee:      00   80483ef:      c7 44 24 14 16 00 00        movl   $0x16,0x14(%esp)   80483f6:      00   80483f7:      c7 44 24 18 21 00 00        movl   $0x21,0x18(%esp)   80483fe:      00   80483ff:      8b 44 24 18                      mov    0x18(%esp),%eax   8048403:      89 44 24 08                    mov    %eax,0x8(%esp)   8048407:      8b 44 24 14                    mov    0x14(%esp),%eax   804840b:      89 44 24 04                    mov    %eax,0x4(%esp)   804840f:      8b 44 24 10                     mov    0x10(%esp),%eax   8048413:      89 04 24                         mov    %eax,(%esp)   8048416:       e8 a9 ff ff ff                     call  80483c4   804841b:      89 44 24 1c                     mov    %eax,0x1c(%esp)   804841f:      b8 04 85 04 08                 mov    $0x8048504,%eax   8048424:      8b 54 24 1c                     mov    0x1c(%esp),%edx   8048428:      89 54 24 04                     mov    %edx,0x4(%esp)   804842c:      89 04 24                         mov    %eax,(%esp)   804842f:      e8 c0 fe ff ff                     call   80482f4   8048434:      b8 00 00 00 00              mov    $0x0,%eax   8048439:       c9                                  leave   804843a:      c3                                  ret   804843b:      90                                 nop   804843c:      90                                 nop   //… 省略部分不相关代码 |

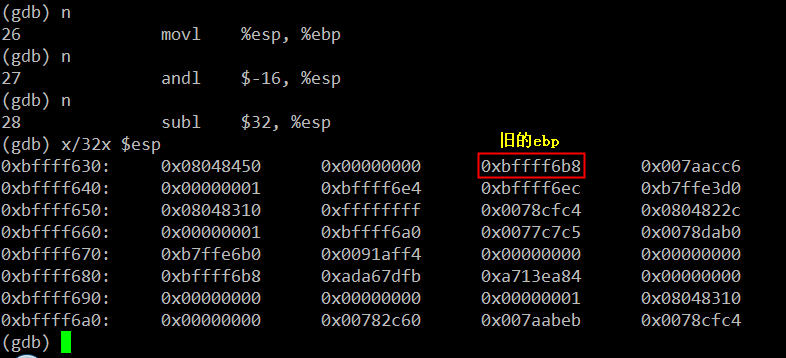
    用GDB调试可执行程序test：

    在main函数第一条指令执行前我们看一下进程test的栈空间布局。因为我们最终的可执行程序是通过glibc库启动的，在main的第一条指令运行前，其实还有很多故事的，这里就不展开了，以后有时间再细究，这里只要记住一点：main函数执行前，其进程空间的栈里已经有了相当多的数据。我的系统里此时栈顶指针esp的值是0xbffff63c，栈基址指针ebp的值0xbffff6b8，指令寄存器eip的值是0x80483de正好是下一条马上即将执行的指令，即main函数内的第一条指令“push %ebp”。那么此时，test进程的栈空间布局大致如下：

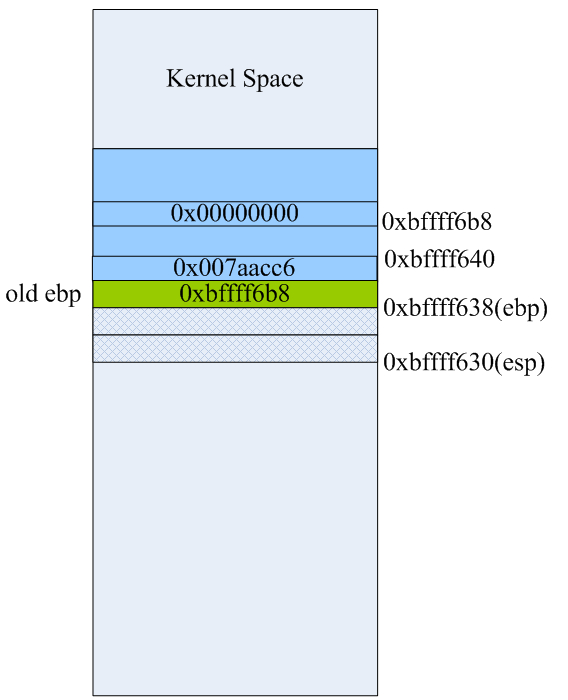
    然后执行如下三条指令：

点击(此处)折叠或打开

1. 25 pushl %ebp         //将原来ebp的值0xbffff6b8如栈，esp自动增长4字节
2. 26 movl %esp, %ebp    //用ebp保存当前时刻esp的值
3. 27 andl $-16, %esp    //内存地址对其，可以忽略不计



   执行完上述三条指令后栈里的数据如上图所示，从0xbffff630到0xbffff638的8字节是为了实现**地址对齐的填充数据**。此时ebp的值0xbffff638，该地址处存放的是ebp原来的值0xbffff6b8。详细布局如下：



   第28条指令“subl  $32,%esp”是在栈上为函数里的本地局部变量预留空间，这里我们看到main主函数有4个int型的变量，理论上说预留16字节空间就可以了，但这里却预留了32字节。GCC编译器在生成汇编代码时，已经考虑到函数调用时其输入参数在栈上的空间预留的问题，这一点我们后面会看到。当第28条指令执行完后栈空间里的数据和布局如下：

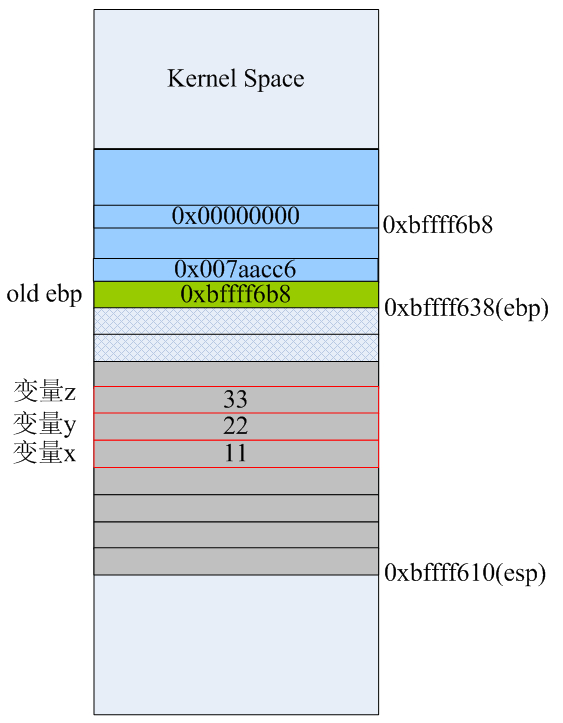


    然后main函数里的变量x，y，z的值放到栈上，就是接下来的三条指令：

点击(此处)折叠或打开

1. 29 movl $11, 16(%esp)
2. 30 movl $22, 20(%esp)
3. 31 movl $33, 24(%esp)

   这是三条寄存器间接寻址指令，将立即数11，22，33分别放到esp寄存器所指向的地址0xbffff610向高位分别偏移16、20、24个字节处的内存单元里，最后结果如下：

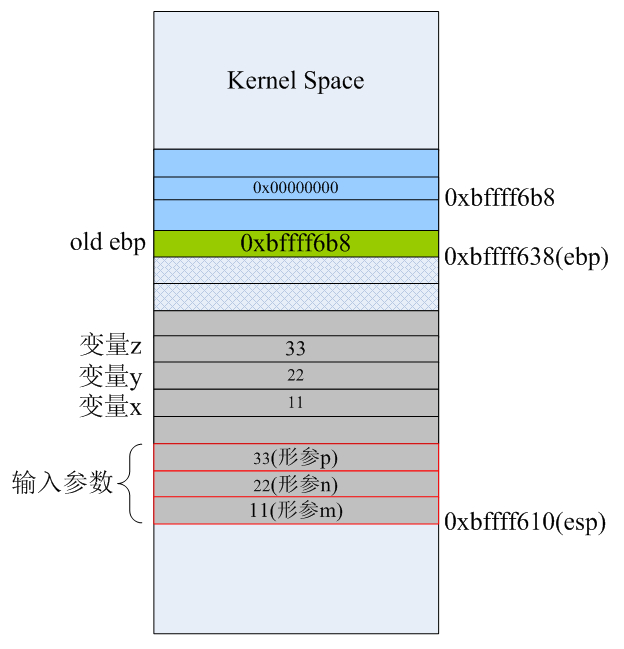


   注意：这三条指令并没有改变esp寄存器的值。

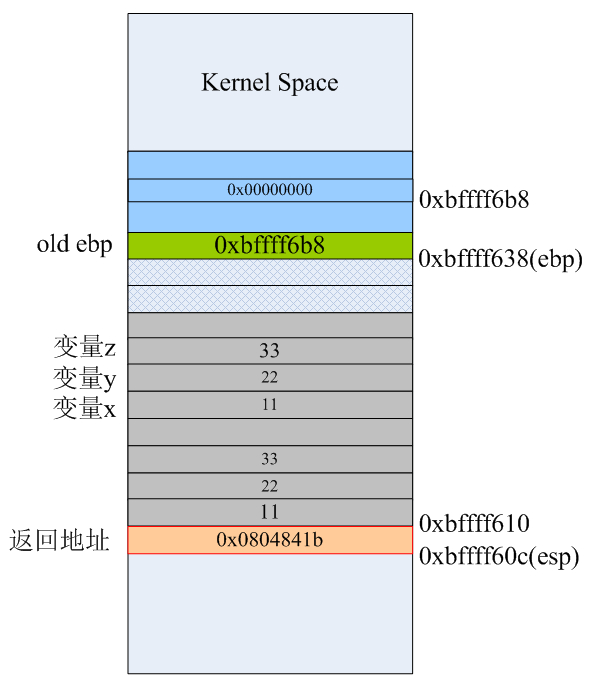
   接下来main函数里就要为了调用foo1函数而做准备了。由于mov指令的两个操作数不能都是内存地址，所以要将x，y和z的值传递给foo1函数，则必须借助通用寄存器来完成，这里我们看到eax承担了这样的任务：

点击(此处)折叠或打开

1. 32 movl 24(%esp), %eax
2. 33 movl %eax, 8(%esp)
3. 34 movl 20(%esp), %eax
4. 35 movl %eax, 4(%esp)
5. 36 movl 16(%esp), %eax
6. 37 movl %eax, (%esp)



    当foo1函数所需要的所有输入参数都已经按正确的顺序入栈后，紧接着就需要调用call指令来执行foo1函数的代码了。前面的博文说过，call指令执行时分两步：首先会将call指令的下一条指令(movl  %eax,28(%esp))的地址(0x0804841b)压入栈，然后跳转到函数foo1入口处开始执行。当第38条指令“call foo1”执行完后，栈空间布局如下：

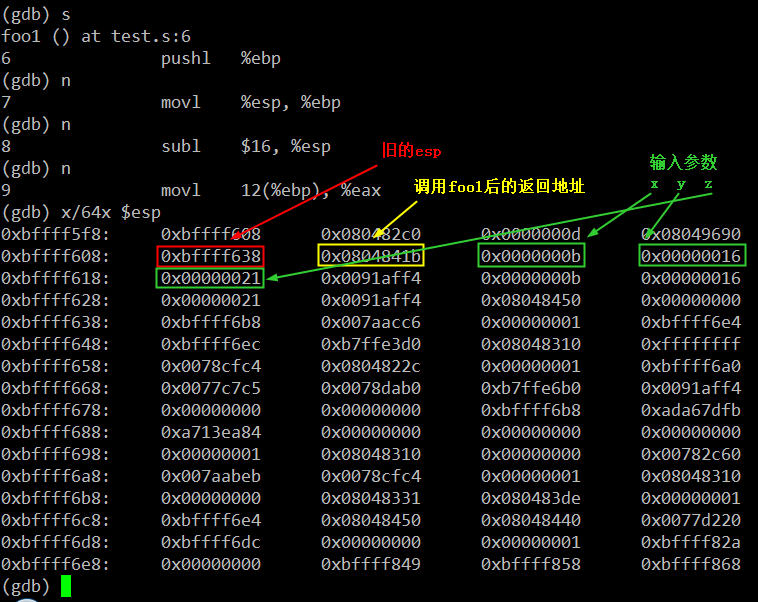


   call指令自动将下一条要执行的指令的地址0x0804841b压入栈，栈顶指针esp自动向低地址处“增长”4字节。所以，我们以前在C语言里所说的函数返回地址，应该理解为：当被调用函数执行完之后要返回到它的调用函数里下一条马上要执行的代码的地址。为了便于观察，我们把foo1函数最后生成指令再列出来：

点击(此处)折叠或打开

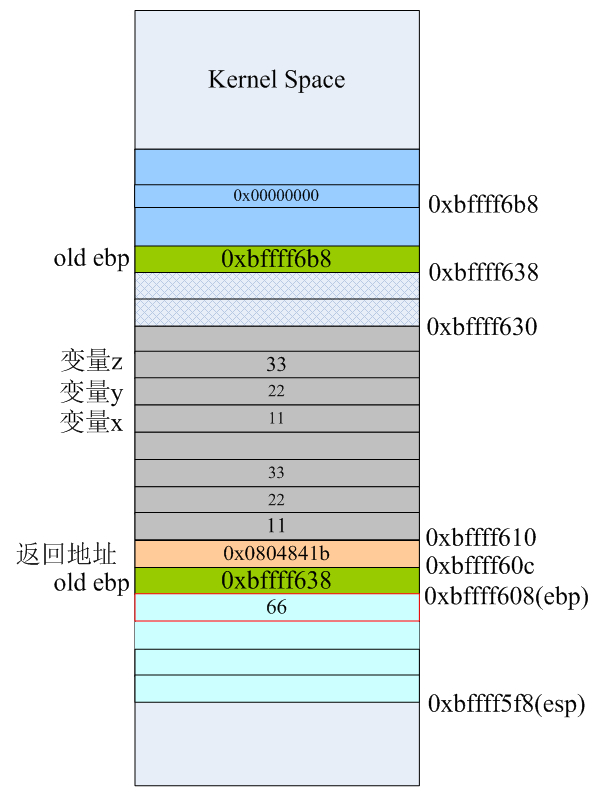
1. 3 .globl foo1
2. 4           .type foo1, @function
3. 5 foo1:
4. 6           pushl %ebp
5. 7           movl %esp, %ebp
6. 8           subl $16, %esp
7. 9           movl 12(%ebp), %eax
8. 10          movl 8(%ebp), %edx
9. 11          leal (%edx,%eax), %eax
10. 12          addl 16(%ebp), %eax
11. 13          movl %eax, -4(%ebp)
12. 14          movl -4(%ebp), %eax
13. 15          leave
14. 16          ret
15. 17          .size foo1, .-foo1

    进入到foo1函数里，开始执行该函数里的指令。当执行完第6、7、8条指令后，栈里的数据如下。这三条指令就是汇编层面函数的“序幕”，分别是保存ebp到栈，让ebp指向当前栈顶，然后为函数里的局部变量预留空间：

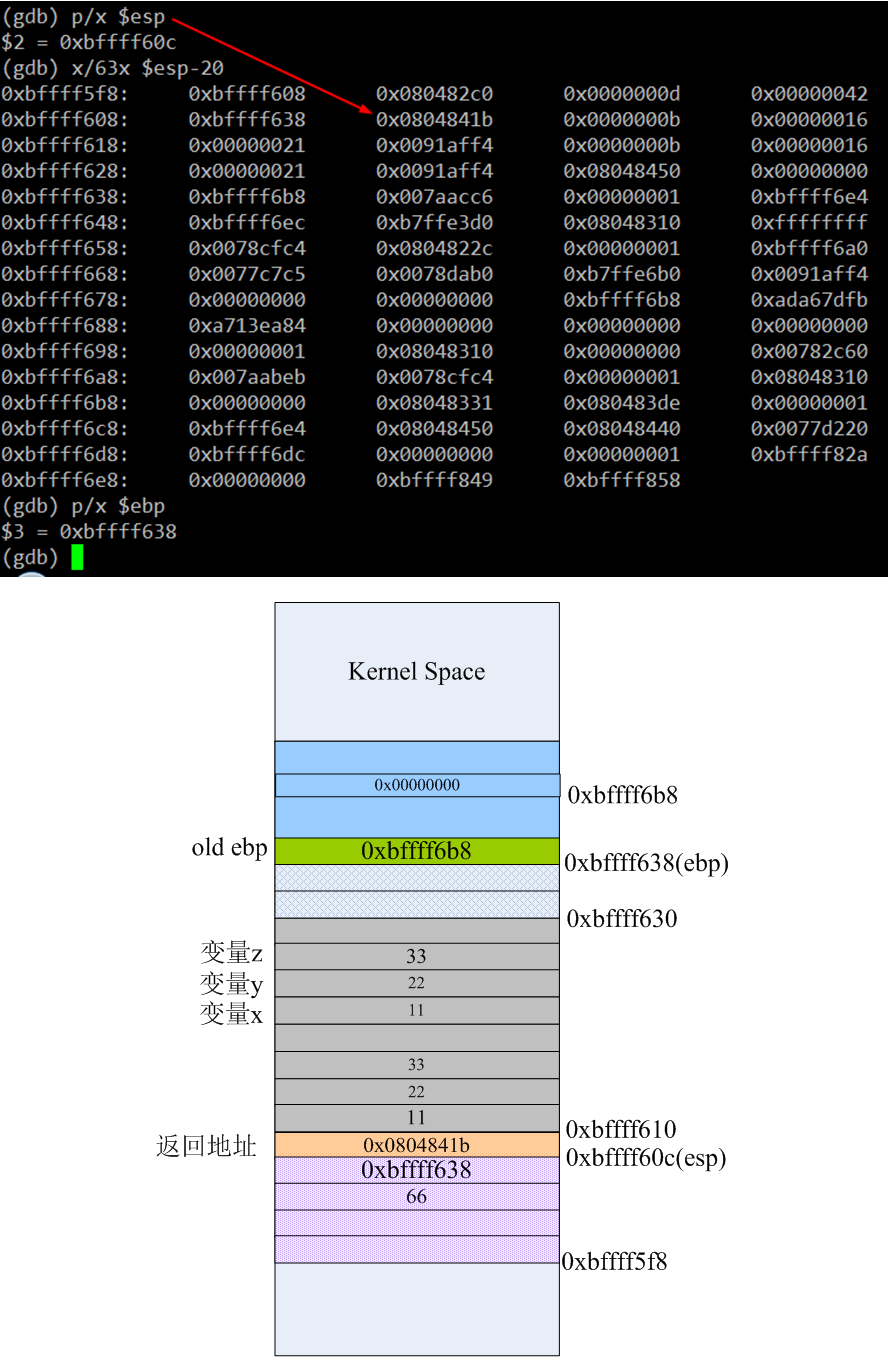


   接下来第9和第10条指令，也并没有改变栈上的任何数据，而是将函数输入参数列表中的的x和y的值分别转载到eax和edx寄存器，和main函数刚开始时做的事情一样。此时eax=22、edx=11。然后用了一条leaf指令完成x和y的加法运算，并将运算结果存在eax里。第12条指令“addl 16(%ebp), %eax”将第三个输入参数p的值，这里是实参z的值为33，同样用寄存器间接寻址模式累加到eax里。此时eax=11+22+33=66就是我们最终要得计算结果。

   因为我们foo1()函数的C代码中，最终计算结果是保存到foo1()里的局部变量x里，最后用return语句将x的值通过eax寄存器返回到mian函数里，所以我们看到接下来的第13、14条指令有些“多此一举”。这足以说明gcc人家还是相当严谨的，C源代码的函数里如果有给局部变量赋值的语句，生成汇编代码时确实会在栈上为本地变量预留的空间里的正确位置为其赋值。当然gcc还有不同级别的优化技术来提高程序的执行效率，这个不属于本文所讨论的东西。让我们继续，当第13、14条指令执行完后，栈布局如下：



   将ebp-4的地址处0xbffff604(其实就是foo1()里的第一个局部参数x的地址)的值设置为66，然后再将该值复制到eax寄存器里，等会儿在main函数里就可以通过eax寄存器来获取最终的计算结果。当第15条指令leave执行完后，栈空间的数据和布局如下：



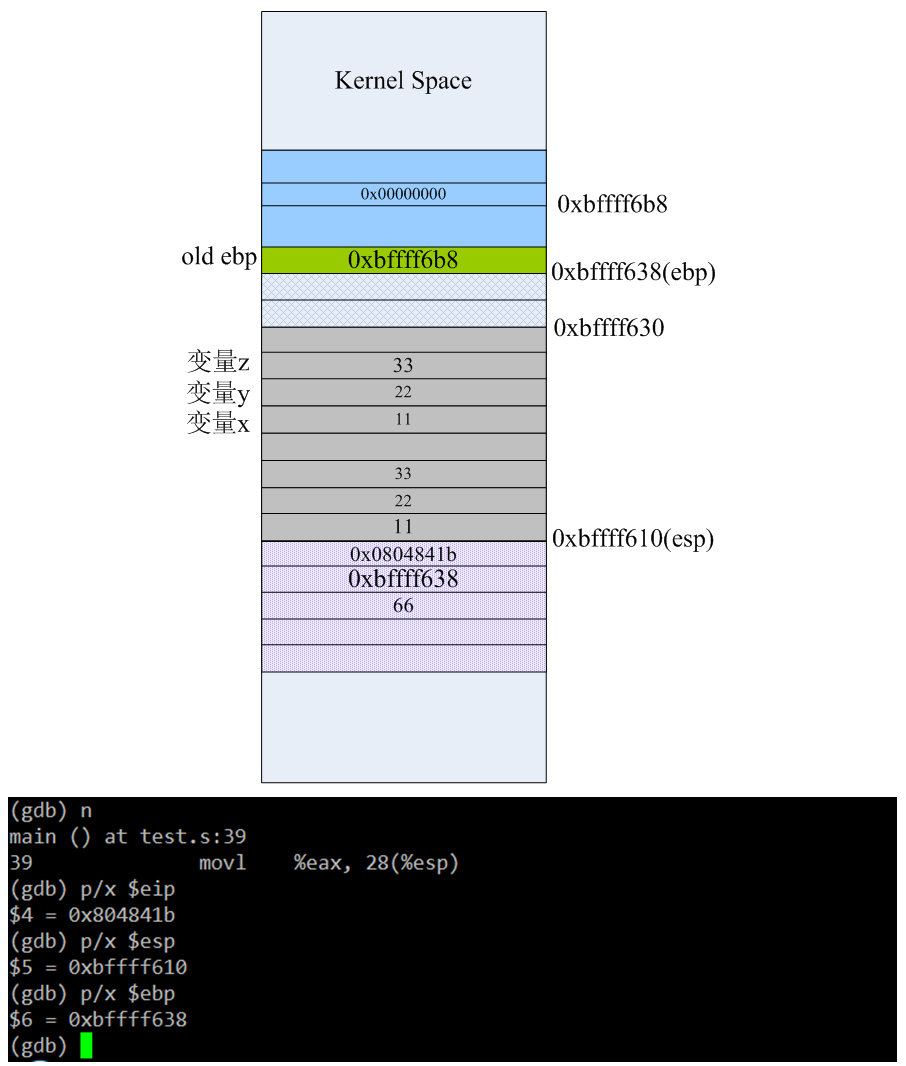
    我们发现，虽然栈顶从0xbffff5f8移动到0xbffff60c了，但栈上的数据依然存在。也就是说，此时你通过esp-8依旧可以访问foo1函数里的局部变量x的值。当然，这也是说得通的，因为函数此时还没有返回。我们看栈布局可以知道当前的栈顶0xbffff60c处存放的是下一条即将执行的指令的地址，对照反汇编结果可以看到这正是main函数里的第18条指令(在整个汇编源文件test.s里的行号是39)“movl  %eax, 28(%esp)”。leave指令其实完成了两个任务：  
   1、将栈上为函数预留的空间“收回”；  
   2、恢复ebp；

   也就是说leave指令等价于下面两条指令，你将leave替换成它们编译运行，结果还是对的：

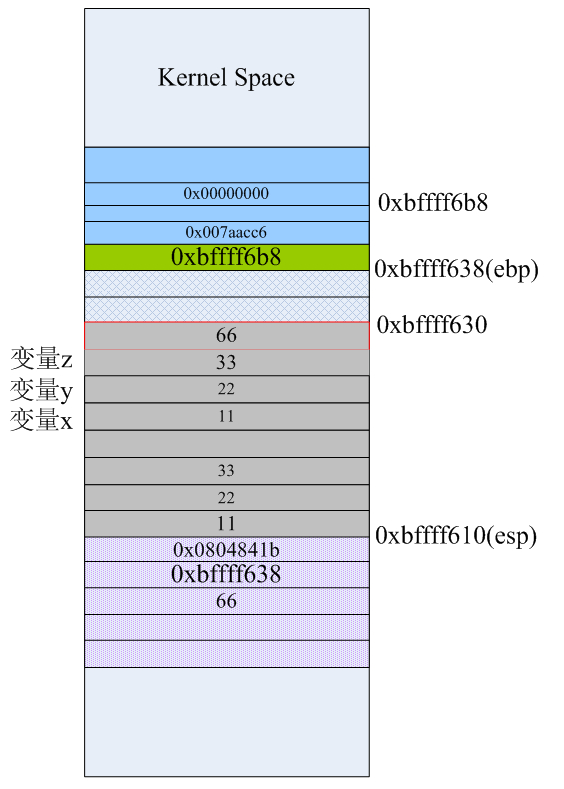
点击(此处)折叠或打开

1. movl %ebp,%esp
2. popl %ebp

   前面我们也说过，ret指令会自动到栈上去pop数据，相当于执行了“popl %eip”，会使esp增大4字节。所以当执行完第16条指令ret后，esp从0xbffff60c增长到0xbffff610处，栈空间结构如下：



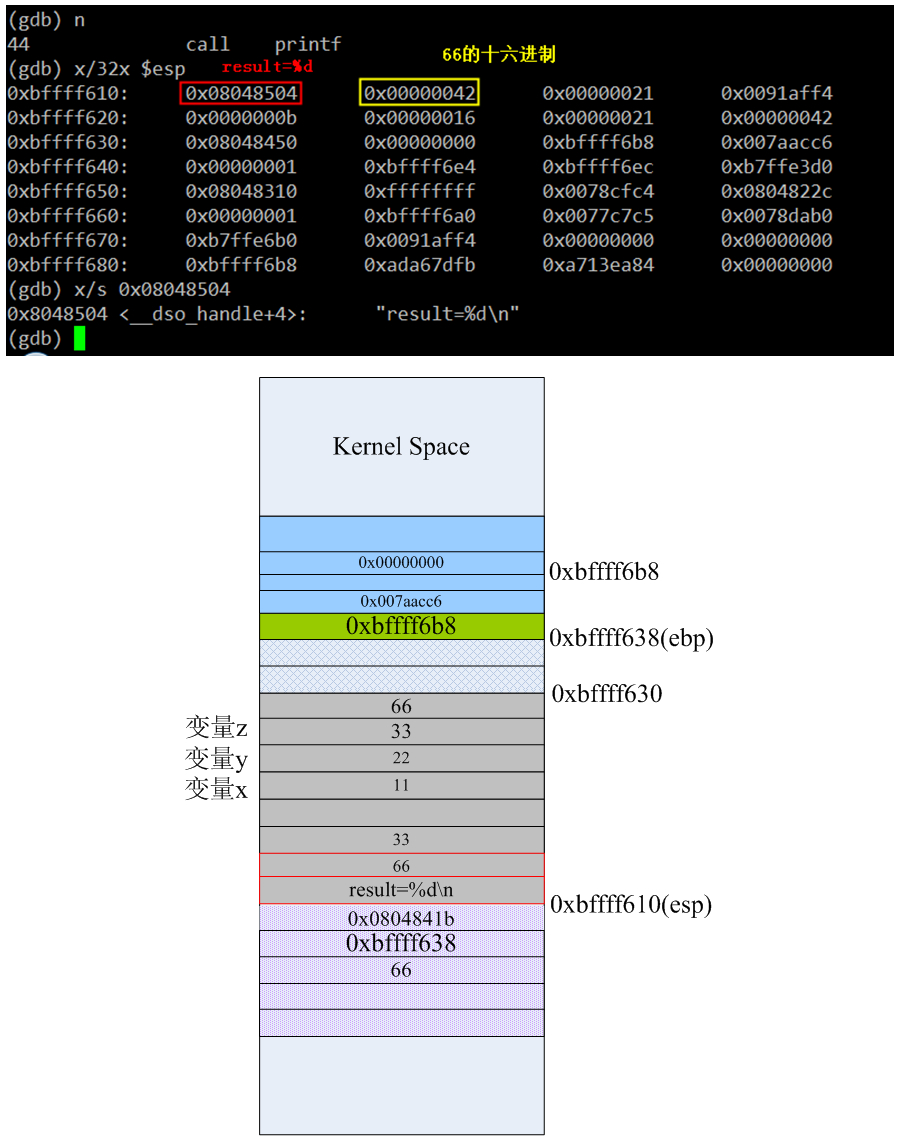
   现在已经从foo1里返回了，但是由于还没执行任何push操作，栈顶“上部”的数据依旧还是可以访问到了，即esp-12的值就是foo1里的局部变量x的值、esp-4的值就是函数的返回地址，当执行第39条指令“movl %eax，28(%esp)”后栈布局变成下面的样子：



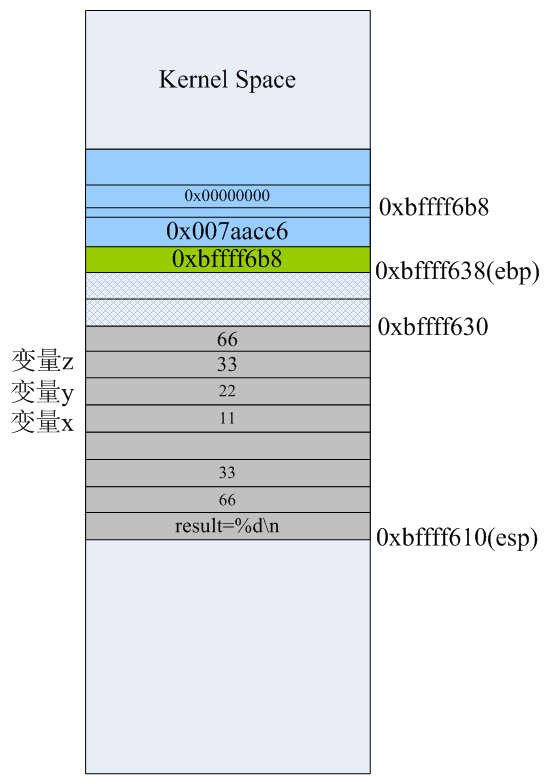
   第39条指令就相当于给main里的result变量赋值66，如上红线标注的地方。接下来main函数里要执行printf("result=%d\n",result)语句了，而printf又是C库的一个常用的输出函数，这里就又会像前面调用foo1那样，初始化栈，然后用“call printf的地址”来调用C函数。当40～43这4条指令执行完后，栈里的数据如下：

点击(此处)折叠或打开

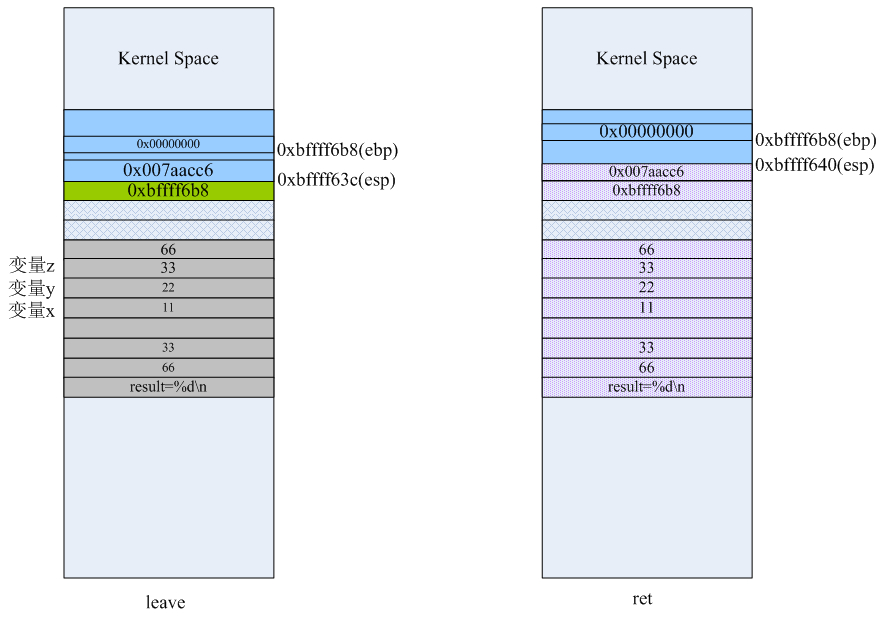
1. 40 movl $.LC0, %eax
2. 41 movl 28(%esp), %edx
3. 42 movl %edx, 4(%esp)
4. 43 movl %eax, (%esp)



   上图为了方便理解，将栈顶的0x08048504替换了成字符串“result=%d\n”，但进程实际运行时此时栈顶esp的值是字符串所在的内存地址。当第44条指令执行完后，栈布局如下：



   由于此时栈已经用来调用printf了，所以栈顶0xbffff610“以上”部分的空间里就找不到foo1的任何影子了。最后在main函数里，当第46、47条指令执行完后栈的布局分别是：

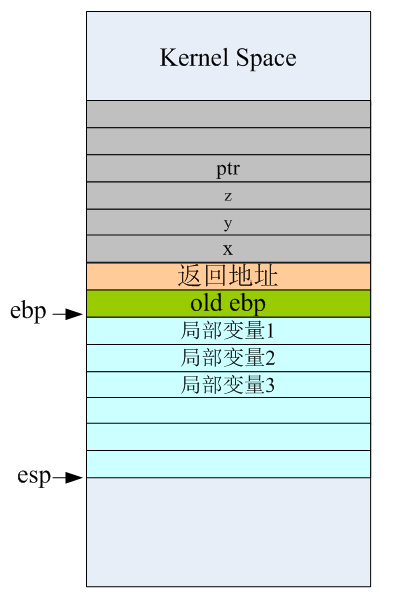


    当main函数里的ret执行完，其实是返回到了C库里继续执行剩下的清理工作。  
   所以，最后关于C的函数调用，我们可以总结一下：  
**1、函数输入参数的入栈顺序是函数原型中形参从右至左的原则；**  
**2、汇编语言里调用函数通常情况下都用call指令来完成；  
   3、汇编语言里的函数大部分情况下都符合以下的函数模板：**

点击(此处)折叠或打开

1. .globl fun\_name
2. .type fun\_name, @function
3. fun\_name:
4. pushl %ebp
5. movl %esp, %ebp
6. <函数主体代码>
7. leave
8. ret

   如果我们有个函数原型：int funtest(int x,int y int z char\* ptr)，在汇编层面，当调用它时栈的布局结构一般是下面这个样子：



而有些资料上将ebp指向函数返回地址的地方，这是不对的。正常情况下应该是ebp指向old ebp才对，这样函数末尾的leave和ret指令才可以正常工作。

<https://www.cnblogs.com/wanghuaijun/p/6801329.html>

[proc文件系统探索之proc根目录下的文件 一](http://www.kerneltravel.net/?p=285) cmdline

========================================

在proc根目录下，以数字命名的目录表示当前一个运行的进程，目录名为进程的pid。其内的目录和文件给出了一些关于进程的信息。

1. ywx@ywx:/proc/1500$ ls
2. ls: cannot read symbolic link cwd: Permission denied
3. ls: cannot read symbolic link root: Permission denied
4. ls: cannot read symbolic link exe: Permission denied
5. attr   cpuset latency   mountstats   sched      status
6. auxv   cwd         limits    net          schedstat  syscall
7. cgroup   environ       loginuid  oom\_adj      sessionid  task
8. clear\_refs              exe       maps         oom\_score  smaps      wchan
9. cmdline   fd            mem       pagemap      stack
10. comm      fdinfo        mountinfo personality  stat
11. coredump\_filter         io        mounts       root       statm

我们可以看到该目录下有这么些文件。其中attr、fd、fdinfo、task为目录，cwd、root为指向目录的链接，exe为指向文件的链接，其余为一般文件。对于一些文件或目录的权限(查看或者修改的权限)是该进程的创建者才有，例如auxv等文件或目录只有创建该进程的用户才有查看或修改的权限，而其他一些文件则对所有用户可读权限。关于这些文件或目录的权限，我们可以在内核中找到(fs/proc/base.c， tid\_base\_stuff数组)。

在2.6.35源码fs/proc/base.c中;<http://lxr.linux.no/linux+v2.6.35/fs/proc/base.c>

1. static const struct pid\_entry tid\_base\_stuff[] = {
2. 2907 DIR("fd", S\_IRUSR|S\_IXUSR, proc\_fd\_inode\_operations, proc\_fd\_operations),
3. 2908 DIR("fdinfo", S\_IRUSR|S\_IXUSR, proc\_fdinfo\_inode\_operations, proc\_fdinfo\_operations),
4. 2909 REG("environ", S\_IRUSR, proc\_environ\_operations),
5. 2910 INF("auxv", S\_IRUSR, proc\_pid\_auxv),
6. 2911 ONE("status", S\_IRUGO, proc\_pid\_status),
7. 2912 ONE("personality", S\_IRUSR, proc\_pid\_personality),
8. 2913 INF("limits", S\_IRUSR, proc\_pid\_limits),
9. 2914#ifdef CONFIG\_SCHED\_DEBUG
10. 2915 REG("sched", S\_IRUGO|S\_IWUSR, proc\_pid\_sched\_operations),
11. 2916#endif
12. 2917 REG("comm", S\_IRUGO|S\_IWUSR, proc\_pid\_set\_comm\_operations),
13. 2918#ifdef CONFIG\_HAVE\_ARCH\_TRACEHOOK
14. 2919 INF("syscall", S\_IRUSR, proc\_pid\_syscall),
15. 2920#endif
16. 2921 INF("cmdline", S\_IRUGO, proc\_pid\_cmdline),
17. 2922 ONE("stat", S\_IRUGO, proc\_tid\_stat),
18. 2923 ONE("statm", S\_IRUGO, proc\_pid\_statm),
19. 2924 REG("maps", S\_IRUGO, proc\_maps\_operations),
20. 2925#ifdef CONFIG\_NUMA
21. 2926 REG("numa\_maps", S\_IRUGO, proc\_numa\_maps\_operations),
22. 2927#endif
23. 2928 REG("mem", S\_IRUSR|S\_IWUSR, proc\_mem\_operations),
24. 2929 LNK("cwd", proc\_cwd\_link),
25. 2930 LNK("root", proc\_root\_link),
26. 2931 LNK("exe", proc\_exe\_link),
27. 2932 REG("mounts", S\_IRUGO, proc\_mounts\_operations),
28. 2933 REG("mountinfo", S\_IRUGO, proc\_mountinfo\_operations),
29. 2934#ifdef CONFIG\_PROC\_PAGE\_MONITOR
30. 2935 REG("clear\_refs", S\_IWUSR, proc\_clear\_refs\_operations),
31. 2936 REG("smaps", S\_IRUGO, proc\_smaps\_operations),
32. 2937 REG("pagemap", S\_IRUSR, proc\_pagemap\_operations),
33. 2938#endif
34. 2939#ifdef CONFIG\_SECURITY
35. 2940 DIR("attr", S\_IRUGO|S\_IXUGO, proc\_attr\_dir\_inode\_operations, proc\_attr\_dir\_operations),
36. 2941#endif
37. 2942#ifdef CONFIG\_KALLSYMS
38. 2943 INF("wchan", S\_IRUGO, proc\_pid\_wchan),
39. 2944#endif
40. 2945#ifdef CONFIG\_STACKTRACE
41. 2946 ONE("stack", S\_IRUSR, proc\_pid\_stack),
42. 2947#endif
43. 2948#ifdef CONFIG\_SCHEDSTATS
44. 2949 INF("schedstat", S\_IRUGO, proc\_pid\_schedstat),
45. 2950#endif
46. 2951#ifdef CONFIG\_LATENCYTOP
47. 2952 REG("latency", S\_IRUGO, proc\_lstats\_operations),
48. 2953#endif
49. 2954#ifdef CONFIG\_PROC\_PID\_CPUSET
50. 2955 REG("cpuset", S\_IRUGO, proc\_cpuset\_operations),
51. 2956#endif
52. 2957#ifdef CONFIG\_CGROUPS
53. 2958 REG("cgroup", S\_IRUGO, proc\_cgroup\_operations),
54. 2959#endif
55. 2960 INF("oom\_score", S\_IRUGO, proc\_oom\_score),
56. 2961 REG("oom\_adj", S\_IRUGO|S\_IWUSR, proc\_oom\_adjust\_operations),
57. 2962#ifdef CONFIG\_AUDITSYSCALL
58. 2963 REG("loginuid", S\_IWUSR|S\_IRUGO, proc\_loginuid\_operations),
59. 2964 REG("sessionid", S\_IRUSR, proc\_sessionid\_operations),
60. 2965#endif
61. 2966#ifdef CONFIG\_FAULT\_INJECTION
62. 2967 REG("make-it-fail", S\_IRUGO|S\_IWUSR, proc\_fault\_inject\_operations),
63. 2968#endif
64. 2969#ifdef CONFIG\_TASK\_IO\_ACCOUNTING
65. 2970 INF("io", S\_IRUGO, proc\_tid\_io\_accounting),
66. 2971#endif
67. 2972};

下面来详细每一个文件和目录的作用。

1.cmdline文件

该文件包含的是该进程的命令行参数，包括进程的启动路径(argv[0])。也就是说例如你在命令行上运行一个hello程序：

1. 打开一个终端：
2. ywx@ywx:~/desktop$ cat hello.c
3. #include <stdio.h>
4. #include <wait.h>
5. int main()
6. {
7. int i=0;
8. for(i=0;i<100;i++)
9. {
10. printf("hello world\n");
11. sleep(2);
12. }
13. return 0;
14. }
15. ywx@ywx:~/desktop$ gcc hello.c -o hello
16. ywx@ywx:~/desktop$ ./hello one two
17. hello world
18. hello world

打开另一个终端，查看当前进程的进程号：得知进程号为

ywx@ywx:~/Desktop$ ps -A | grep hello

 2752 pts/0    00:00:00 hello

然后进入/proc/2752进程下面：

ywx@ywx:~/Desktop$ cd /proc/2752

ywx@ywx:/proc/2752$ cat cmdline

./helloonetwoywx@ywx:/proc/2752$

可以看到cmdline里的内容为 “./helloonetwo”，正是命令行的参数。可能你会疑问为什么参数没有分开？？那是因为cat欺骗了你。我们可以将cmdline复制到desktop下，然后用vim查看发现是这样的：

1. ./hello^@one^@two^@

也就是说，实际上每个参数之间是有东西隔开的，只不过cat将其忽略而已，而vim可以给你标识出东西，但vim本身不显示罢了。我们可以通过编程读取该文件。下面给出程序。

我们一个字符一个字符的读取文件内容直到文件结束，在读取每一个字符的时候，打印其字符和对应的数值。

1. ywx@ywx:~/desktop$ cat readcmd.c
2. #include <stdio.h>// std io fopen() snprintf() feof() perrof()
3. int main(int argc,char \*argv[])
4. {
5. FILE \*fp;
6. char path[80];
7. unsigned char ch;
8. snprintf(path,80,"/home/ywx/desktop/cmdline");//将cmdline中内容写入path数组中，并在path的结尾添 加字符结束符"\0"
9. if((fp=fopen(path,"r")) == NULL)
10. {
11. perror("fopen");
12. return 0;
13. }
14. while(!feof(fp)) //判断当前操作位置是否为文件的末尾，如果是，返回一个非零值
15. {
16. ch=fgetc(fp);//从stream流中读取一个字符，操作位置向下移动一个
17. printf("%c %d\n",ch,ch);
18. }
19. fclose(fp);
20. return 0;
21. }

ywx@ywx:~/Desktop$ gcc readcmd.c -o readcmd

ywx@ywx:~/Desktop$ ./readcmd

. 46

/ 47

h 104

e 101

l 108

l 108

o 111

 0

o 111

n 110

e 101

 0

t 116

w 119

o 111

 0

� 255

由此我们可以看到并非是每个参数之间没有间隔，而是以字符"\0"作为间隔。所以如果我们在某一个程序中想读取进程的命令行参数，我们只需要知道该进程的pid，然后进入proc文件系统的该pid对应的目录下，编程读写读取cmdline文件就可以了。

<http://blog.chinaunix.net/uid-22666248-id-3040364.html>

2. cwd目录链接

该目录链接指向该进程运行的当前路径。该符号链接虽然使用ls命令查看其权限是对所有用户都有权限，但实际上是只用启动该进程的用户才具有读写的权限，其他用户不具有一切权限。该链接指向该进程运行的当前路径看，例如我们在用户目录下启动该进程，那么cwd就指向用户目录。

使用上一节使用的hello.c程序 <http://blog.chinaunix.net/space.php?uid=22666248&do=blog&id=3040364>

1. ywx@ywx:~/Desktop/proc$ ls
2. hello  hello.c  readcmd  readcmd.c
3. ywx@ywx:~/Desktop/proc$ ./hello one two
4. hello world
5. ywx@ywx:~/desktop$ ps -A | grep hello
6. 3593 pts/1 00:00:00 hello
7. ywx@ywx:~/desktop$ cd /proc/3593
8. ywx@ywx:/proc/3593$ cd cwd
9. ywx@ywx:/proc/3593/cwd$ ls
10. hello hello.c readcmd readcmd.c hello这个进程程序在/home/ywx/desktop/proc/下面
11. ywx@ywx:/proc/3593/cwd$

3.environ文件

包含该进程运行的环境变量。我们常用一些环境变量都包含在该文件中，例如PATH、HOME、PWD等。所以如果我们想在一个进程中获取这些环境变量而又不想使用getenv(),getpwd等函数的时候，我们就可以直接读取该进程的该文件以直接获得环境变量。下面是一个程序实现这个过程：

1. ywx@ywx:~/Desktop/proc$ cat environ.c
2. #include <stdio.h>
3. #include <unistd.h> //unix stdard getpid()
4. #include <string.h> //[strstr()](http://www.cnljh.com/clib/string/strstr.html)
5. int main(int argc, char \*argv[])
6. {
7. FILE \*fp;
8. int pid;
9. char path[80];
10. int i = 0;
11. unsigned char buff[1024\*3];
12. unsigned char \*p = NULL;
13. unsigned char ch;
14. if(argc < 2)
15. {
16. printf("no argument\n");
17. return 0;
18. }
19. pid = getpid();
20. snprintf(path,80,"/proc/%d/environ",pid);//格式输入数据
21. if((fp = fopen(path,"r")) == NULL)
22. {
23. perror("fopen");
24. return 0;
25. }
26. while(!feof(fp))
27. {
28. if((ch=fgetc(fp))!='\0')//从environ文件中得到数据，存储在buff中
29. {
30. buff[i]=ch;
31. i++;
32. continue;
33. }
34. buff[i]='\0';
35. //从字符串buffe中寻找argv[1]第一次出现的位置，没找到，返回NULL
36. if((p=strstr(buff,argv[1])) != NULL)//not find return NULL
37. {
38. printf("%s\n",p+strlen(argv[1]+1));//找到，打印信息
39. }
40. i = 0;
41. memset(buff,'\0',1024\*3);
42. }
43. fclose(fp);
44. return 0;
45. }

ywx@ywx:~/desktop/proc$ gcc environ.c -o environ

ywx@ywx:~/desktop/proc$ ./environ HOME

E=/home/ywx

ywx@ywx:~/desktop/proc$ ./environ PWD

D=/home/ywx/desktop/proc

D=/proc/3542

ywx@ywx:~/desktop/proc$

4. exe链接文件

指向该进程相应的可执行文件。从这里我们可以看到该进程的可执行文件路径。

我们这里使用cwd目录链接 中hello程序

1. ywx@ywx:/proc$ ps -A | grep hello
2. 4251 pts/1 00:00:00 hello
3. ywx@ywx:/proc$ sudo ls -l /proc/4251/exe
4. lrwxrwxrwx 1 ywx ywx 0 2011-12-13 11:08 /proc/4251/exe -> /home/ywx/Desktop/proc/hello
5. ywx@ywx:/proc$

5.maps文件和smaps文件

maps文件是可执行文件或者库文件对应的内存映像，而smpas文件显示的是该进程这些可执行文件或库文件内存映像在内存中的大小等信息。

1. ywx@ywx:/proc/1500$ sudo cat maps | more
2. 00110000-00112000 r-xp 00000000 08:01 1575226 /usr/lib/libgmodule-2.0.so.0.2600.1 代码段
3. 00112000-00113000 r--p 00002000 08:01 1575226 /usr/lib/libgmodule-2.0.so.0.2600.1 看不出来
4. 00113000-00114000 rw-p 00003000 08:01 1575226 /usr/lib/libgmodule-2.0.so.0.2600.1 数据段
5. 00114000-00117000 r-xp 00000000 08:01 1575228 /usr/lib/libgthread-2.0.so.0.2600.1
6. 00117000-00118000 r--p 00003000 08:01 1575228 /usr/lib/libgthread-2.0.so.0.2600.1
7. 00118000-00119000 rw-p 00004000 08:01 1575228 /usr/lib/libgthread-2.0.so.0.2600.1
8. 00119000-00131000 r-xp 00000000 08:01 1576735 /usr/lib/libxcb.so.1.1.0
9. 00131000-00132000 r--p 00017000 08:01 1576735 /usr/lib/libxcb.so.1.1.0
10. 00132000-00133000 rw-p 00018000 08:01 1576735 /usr/lib/libxcb.so.1.1.0

我们可以看到pid等于1500的进程的内存映像。其中从00110000-00112000 是  /libgmoudle-2.0.so.2600.1的内存映像 ，骑在虚拟内存中的偏移是0x000000 ，该文件的inode是1575226，其访问标志是 r-xp，即可读、执行、私有的。我们可以猜出此为/libgmoudle的程序段。其中00112000-00113000  也是内存映射，r--p 表示：可读、私有的。其中00113000-00114000  也是内存映射，rw-p 是数据段。

08:01 表示的是文件所在设备的设备号  08 表示是主设备号，01 表示次设备号。这里的08 cat /proc/devices 显示是表示 8 sd   设备类型标号参考/linux/Documention/devices.txt文件。  所以我们得知此设备是/dev/sd1 。

查看smaps的内容

1. ywx@ywx:/proc/1500$ sudo cat smaps | more
2. 00110000-00112000 r-xp 00000000 08:01 1575226 /usr/lib/libgmodule-2.0.so.0.2600.1
3. Size: 8 kB
4. Rss: 8 kB
5. Pss: 0 kB
6. Shared\_Clean: 8 kB
7. Shared\_Dirty: 0 kB
8. Private\_Clean: 0 kB
9. Private\_Dirty: 0 kB
10. Referenced: 8 kB
11. Swap: 0 kB
12. KernelPageSize: 4 kB
13. MMUPageSize: 4 kB

我们可以看到其显示的信息比maps文件更详细。对于内存映像的每一个段它都列出该段的详细信息，如大小等。我们可以结合这两个文件获取该进程的完整的内存映像信息。

<http://blog.chinaunix.net/uid-22666248-id-3040440.html>

1. **char** buff[64] = {0x00};
3. sprintf(buff,"cat /proc/%d/maps", getpid());
5. system((**const** **char**\*) buff);

程序中执行shell 命令： cat /proc/%d/maps

产生段错误就是访问了错误的内存段，一般是你没有权限，或者根本就不存在对应的物理内存，尤其常见的是访问0地址。

一般来说，段错误就是指访问的内存超出了系统所给这个程序的内存空间，通常这个值是由gdtr来保存的，它是一个48位的寄存器，其中的32位是保存由它指向的gdt表，后13位保存相应于gdt的下标，最后3位包括了程序是否在内存中以及程序在cpu中的运行级别，指向的gdt是以64位为一个单位的表，在这张表中就保存着程序运行的代码段、数据段的起始地址、与此相应的段限和页面交换、程序运行级别还有内存粒度等等的信息。一旦一个程序发生了越界访问，cpu就会产生相应的异常保护，于是segmentation fault就出现了.

在编程中以下几类做法容易导致段错误，基本是是错误地使用指针引起的：

1)访问系统数据区，尤其是往系统保护的内存地址写数据，最常见的就是给一个指针以0地址；

2)内存越界(数组越界，变量类型不一致等) 访问到不属于你的内存区域。

解决方法

我们在用C/C++语言写程序的时侯，内存管理的绝大部分工作都是需要我们来做的。实际上，内存管理是一个比较繁琐的工作，无论你多高明，经验多丰富， 难免会在此处犯些小错误，而通常这些错误又是那么的浅显而易于消除。但是手工“除虫”（debug），往往是效率低下且让人厌烦的，本文将就“段错误”这个 内存访问越界的错误谈谈如何快速定位这些“段错误”的语句。

下面将就以下的一个存在段错误的程序介绍几种调试方法：

1 dummy\_function (void)

2 {

3 unsigned char \*ptr = 0x00;

4 \*ptr = 0x00;

5 }

6

7 int main (void)

8 {

9 dummy\_function ();

10

11 return 0;

12 }

　　作为一个熟练的C/C++程序员，以上代码的bug应该是很清楚的，因为它尝试操作地址为0的内存区域，而这个内存区域通常是不可访问的禁区，当然就会出错了。我们尝试编译运行它:

xiaosuo@gentux test $ ./a.out

段错误

　　果然不出所料，它出错并退出了。

　　1.利用gdb逐步查找段错误:

这种方法也是被大众所熟知并广泛采用的方法，首先我们需要一个带有调试信息的可执行程序，所以我们加上“-g -rdynamic"的参数进行编译，然后用gdb调试运行这个新编译的程序,具体步骤如下:

xiaosuo@gentux test $ gcc -g -rdynamic d.c

xiaosuo@gentux test $ gdb ./a.out

GNU gdb 6.5

Copyright (C) 2006 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are

welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details.

This GDB was configured as "i686-pc-linux-gnu"...Using host libthread\_db library "/lib/libthread\_db.so.1".

(gdb) r

Starting program: /home/xiaosuo/test/a.out

Program received signal SIGSEGV, Segmentation fault.

0x08048524 in dummy\_function () at d.c:4

4 \*ptr = 0x00;

(gdb)

　　好像不用一步步调试我们就找到了出错位置d.c文件的第4行，其实就是如此的简单。

从这里我们还发现进程是由于收到了SIGSEGV信号而结束的。通过进一步的查阅文档(man 7 signal)，我们知道SIGSEGV默认的处理动作是打印“段错误”的出错信息，并产生Core文件，由此我们又产生了方法二。

　　2.分析Core文件：

Core文件是什么呢？

The default action of certain signals is to cause a process to terminate and produce a core dump file, a disk file containing an image of the process's memory at the time of termination. A list of the signals which cause a process to dump core can be found in signal(7).

　　以上资料摘自man page(man 5 core)。不过奇怪了，我的系统上并没有找到core文件。后来，忆起为了减少系统上的拉圾文件的数量（本人有些洁癖，这也是我喜欢Gentoo的原因之一），禁止了core文件的生成，查看了以下果真如此，将系统的core文件的大小限制在512K大小，再试：

xiaosuo@gentux test $ ulimit -c

0

xiaosuo@gentux test $ ulimit -c 1000

xiaosuo@gentux test $ ulimit -c

1000

xiaosuo@gentux test $ ./a.out

段错误 (core dumped)

xiaosuo@gentux test $ ls

a.out core d.c f.c g.c pango.c test\_iconv.c test\_regex.c

　　core文件终于产生了，用gdb调试一下看看吧:

xiaosuo@gentux test $ gdb ./a.out core

GNU gdb 6.5

Copyright (C) 2006 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are

welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details.

This GDB was configured as "i686-pc-linux-gnu"...Using host libthread\_db library "/lib/libthread\_db.so.1".

warning: Can't read pathname for load map: 输入/输出错误.

Reading symbols from /lib/libc.so.6...done.

Loaded symbols for /lib/libc.so.6

Reading symbols from /lib/ld-linux.so.2...done.

Loaded symbols for /lib/ld-linux.so.2

Core was generated by `./a.out'.

Program terminated with signal 11, Segmentation fault.

#0 0x08048524 in dummy\_function () at d.c:4

4 \*ptr = 0x00;

　　还是一步就定位到了错误所在地，佩服一下Linux/Unix系统的此类设计。

接着考虑下去，以前用windows系统下的ie的时侯，有时打开某些网页，会出现“运行时错误”，这个时侯如果恰好你的机器上又装有windows的编译器的话，他会弹出来一个对话框，问你是否进行调试，如果你选择是，编译器将被打开，并进入调试状态，开始调试。

　　Linux下如何做到这些呢？我的大脑飞速地旋转着，有了，让它在SIGSEGV的handler中调用gdb，于是第三个方法又诞生了：

　　3.段错误时启动调试:

#include

#include

#include

#include

void dump(int signo)

{

char buf[1024];

char cmd[1024];

FILE \*fh;

snprintf(buf, sizeof(buf), "/proc/%d/cmdline", getpid());

if(!(fh = fopen(buf, "r")))

exit(0);

if(!fgets(buf, sizeof(buf), fh))

exit(0);

fclose(fh);

if(buf[strlen(buf) - 1] == 'n')

buf[strlen(buf) - 1] = '';

snprintf(cmd, sizeof(cmd), "gdb %s %d", buf, getpid());

system(cmd);

exit(0);

}

void

dummy\_function (void)

{

unsigned char \*ptr = 0x00;

\*ptr = 0x00;

}

int

main (void)

{

signal(SIGSEGV, &dump);

dummy\_function ();

return 0;

}

　　编译运行效果如下：

xiaosuo@gentux test $ gcc -g -rdynamic f.c

xiaosuo@gentux test $ ./a.out

GNU gdb 6.5

Copyright (C) 2006 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are

welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details.

This GDB was configured as "i686-pc-linux-gnu"...Using host libthread\_db library "/lib/libthread\_db.so.1".

Attaching to program: /home/xiaosuo/test/a.out, process 9563

Reading symbols from /lib/libc.so.6...done.

Loaded symbols for /lib/libc.so.6

Reading symbols from /lib/ld-linux.so.2...done.

Loaded symbols for /lib/ld-linux.so.2

0xffffe410 in \_\_kernel\_vsyscall ()

(gdb) bt

#0 0xffffe410 in \_\_kernel\_vsyscall ()

#1 0xb7ee4b53 in waitpid () from /lib/libc.so.6

#2 0xb7e925c9 in strtold\_l () from /lib/libc.so.6

#3 0x08048830 in dump (signo=11) at f.c:22

#4

#5 0x0804884c in dummy\_function () at f.c:31

#6 0x08048886 in main () at f.c:38

　　怎么样？是不是依旧很酷？

　　以上方法都是在系统上有gdb的前提下进行的，如果没有呢？其实glibc为我们提供了此类能够dump栈内容的函数簇，详见 /usr/include/execinfo.h（这些函数都没有提供man page，难怪我们找不到），另外你也可以通过gnu的手册进行学习。

　　4.利用backtrace和objdump进行分析：

　　重写的代码如下:

#include

#include

#include

#include

/\* A dummy function to make the backtrace more interesting. \*/

void

dummy\_function (void)

{

unsigned char \*ptr = 0x00;

\*ptr = 0x00;

}

void dump(int signo)

{

void \*array[10];

size\_t size;

char \*\*strings;

size\_t i;

size = backtrace (array, 10);

strings = backtrace\_symbols (array, size);

printf ("Obtained %zd stack frames.n", size);

for (i = 0; i < size; i++)

printf ("%sn", strings[i]);

free (strings);

exit(0);

}

int

main (void)

{

signal(SIGSEGV, &dump);

dummy\_function ();

return 0;

}

　　编译运行结果如下：

xiaosuo@gentux test $ gcc -g -rdynamic g.c

xiaosuo@gentux test $ ./a.out

Obtained 5 stack frames.

./a.out(dump+0x19) [0x80486c2]

[0xffffe420]

./a.out(main+0x35) [0x804876f]

/lib/libc.so.6(\_\_libc\_start\_main+0xe6) [0xb7e02866]

./a.out [0x8048601]

　　这次你可能有些失望,似乎没能给出足够的信息来标示错误,不急,先看看能分析出来什么吧,用objdump反汇编程序,找到地址0x804876f对应的代码位置:

xiaosuo@gentux test $ objdump -d a.out

8048765: e8 02 fe ff ff call 804856c

　　我们还是找到了在哪个函数(dummy\_function)中出错的,信息已然不是很完整,不过有总比没有好的啊!

　　后记：

　　本文给出了分析"段错误"的几种方法,不要认为这是与孔乙己先生的"回"字四种写法一样的哦,因为每种方法都有其自身的适用范围和适用环境,请酌情使用,或遵医嘱

**https://man.linuxde.net/od**

**od命令**

[文件内容查看](https://man.linuxde.net/sub/%e6%96%87%e4%bb%b6%e5%86%85%e5%ae%b9%e6%9f%a5%e7%9c%8b)

**od命令**用于输出文件的八进制、十六进制或其它格式编码的字节，通常用于显示或查看文件中不能直接显示在终端的字符。

常见的文件为文本文件和二进制文件。此命令主要用来查看保存在二进制文件中的值。比如，程序可能输出大量的数据记录，每个数据是一个单精度浮点数。这些数据记录存放在一个文件中，如果想查看下这个数据，这时候od命令就派上用场了。在我看来，od命令主要用来格式化输出文件数据，即对文件中的数据进行无二义性的解释。不管是IEEE754格式的浮点数还是ASCII码，od命令都能按照需求输出它们的值。

**语法**

od(选项)(参数)

**选项**

-a：此参数的效果和同时指定“-ta”参数相同；

-A：<字码基数>：选择以何种基数计算字码；

-b：此参数的效果和同时指定“-toC”参数相同；

-c：此参数的效果和同时指定“-tC”参数相同；

-d：此参数的效果和同时指定“-tu2”参数相同；

-f：此参数的效果和同时指定“-tfF”参数相同；

-h：此参数的效果和同时指定“-tx2”参数相同；

-i：此参数的效果和同时指定“-td2”参数相同；

-j<字符数目>或--skip-bytes=<字符数目>：略过设置的字符数目；

-l：此参数的效果和同时指定“-td4”参数相同；

-N<字符数目>或--[read](http://man.linuxde.net/read)-bytes=<字符数目>：到设置的字符树目为止；

-o：此参数的效果和同时指定“-to2”参数相同；

-s<字符串字符数>或--strings=<字符串字符数>：只显示符合指定的字符数目的字符串；

-t<输出格式>或--format=<输出格式>：设置输出格式；

-v或--output-duplicates：输出时不省略重复的数据；

-[w](http://man.linuxde.net/w)<每列字符数>或--width=<每列字符数>：设置每列的最大字符数；

-x：此参数的效果和同时指定“-h”参数相同；

--[help](http://man.linuxde.net/help)：在线帮助；

--version：显示版本信息。

1. 在另一个终端，查看进程信息
2. # ps -C a.out -o ppid,pid,stat,cmd
3. PPID PID STAT CMD