有一句话叫做常在河边走，哪有不湿鞋。我们这些研究和开发Android的工程师正应了这句话，相必大家在调试的时候经常会遇到这么个东西吧

\*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\*  
Build fingerprint: 'XXXXXXXXX'  
**pid: 1658, tid: 13086  >>> system\_server <<<**  
signal 11 (SIGSEGV), code 1 (SEGV\_MAPERR), fault addr 64696f7e  
 r0 00000000  r1 00000001  r2 ad12d1e8  r3 7373654d  
 r4 64696f72  r5 00000406  r6 00974130  r7 40d14008  
 r8 4b857b88  r9 4685adb4  10 00974130  fp 4b857ed8  
 ip 00000000  sp 4b857b50  lr afd11108  pc ad115ebc  cpsr 20000030  
 d0  4040000040000000  d1  0000004200000003  
 d2  4e72cd924285e370  d3  00e81fe04b1b64d8  
 d4  3fbc71c7009b64d8  d5  3fe999999999999a  
 d6  4010000000000000  d7  4000000000000000  
 d8  4000000000000000  d9  0000000000000000  
 d10 0000000000000000  d11 0000000000000000  
 d12 0000000000000000  d13 0000000000000000  
 d14 0000000000000000  d15 0000000000000000  
 scr 80000012  
  
         #00  pc 000108d8  /system/lib/libc.so  
         #01  pc 0003724c  /system/lib/libxvi020.so  
         #02  pc 0000ce02  /system/lib/libxvi020.so  
         #03  pc 0000d672  /system/lib/libxvi020.so  
         #04  pc 00010cce  /system/lib/libxvi020.so  
         #05  pc 00004432  /system/lib/libwimax\_jni.so  
         #06  pc 00011e74  /system/lib/libdvm.so  
         #07  pc 0004354a  /system/lib/libdvm.so  
         #08  pc 00017088  /system/lib/libdvm.so  
         #09  pc 0001c210  /system/lib/libdvm.so  
         #10  pc 0001b0f8  /system/lib/libdvm.so  
         #11  pc 00059c24  /system/lib/libdvm.so  
         #12  pc 00059e3c  /system/lib/libdvm.so  
         #13  pc 0004e19e  /system/lib/libdvm.so  
         #14  pc 00011b94  /system/lib/libc.so  
         #15  pc 0001173c  /system/lib/libc.so  
  
code around pc:  
ad115e9c 4620eddc bf00bd70 0001736e 0001734e  
ad115eac 4605b570 447c4c0a f7f44620 e006edc8  
ad115ebc 42ab68e3 68a0d103 f7f42122 6864edd2  
ad115ecc d1f52c00 44784803 edbef7f4 bf00bd70  
ad115edc 00017332 00017312 2100b51f 46682210  
  
code around lr:  
afd110e8 e2166903 1a000018 e5945000 e1a02004  
afd110f8 e2055a02 e1a00005 e3851001 ebffed92  
afd11108 e3500000 13856002 1a000001 ea000009  
afd11118 ebfffe50 e1a01004 e1a00006 ebffed92  
afd11128 e1a01005 e1550000 e1a02006 e3a03000

stack:  
    4b857b10  40e43be8    
    4b857b14  00857280    
    4b857b18  00000000    
    4b857b1c  034e8968    
    4b857b20  ad118ce9  /system/lib/libnativehelper.so  
    4b857b24  00000002    
    4b857b28  00000406

    .....

    初一看到直接崩溃，我的个妈，这么一大坨，啥玩意啊，完全搞不懂，没有头绪撒。这NND Android出了问题，冒出这么大堆的地址，搞毛呢。

   其实， 我可以很负责任的告诉你，兄弟这么多其实我也看不懂，在我眼里除了红色和蓝色的地方，其他全是废话。因为除了这两个地方意外，其他的我确实看不懂，一堆线性地址，外加寄存器，跟火星文一样。

   算了，废话不多少，直接分析:

   1 红色的地方，是让我们确认问题到底发生在那个线程中，是主线程还是子线程，这个的判断依据是:如果PID和TID相同，恭喜你问题出在父亲这边，看样子问 题还比较好办。如果PID和TID不相同，那么您悲剧了，问题出在子线程中。根据我的经验，100的tombstone中，问题出现在父线程的概率，我还 从来没发现，尤其是什么system\_server, zygote这些，基本都是儿子的问题。唉，谁叫父亲英雄，儿狗熊呢。。。

  2 确认了问题的基本点，下面就是开始看蓝色的部分了。

    这个蓝色不分是从#00-->#XX 表面上看是从上往下的增长，其实坑爹啊，它所显示的程序的执行流程，恰恰是从下往上的也就是实际的执行顺序是#XX-->#00。

   所以一开始我们的目的是分析第一个出现问题的动态连接库  
   #15  pc 0001173c  /system/lib/libc.so

  a 首先按图索骥，找到你的目标，libc.so这个一般会在你编译完的目标目录下，也就是**out/target/product/your\_pro/system/lib**这个目录下。

  b 一般你如果用的是还像样的linux系统，会有个地址解析的命令addr2line，负责解析动态连接库的（如果你是悲剧，没有这个命令，那好吧， google早就预料到会有象你这样的悲剧人，他们在prebuilt/linux-x86/toolchain/arm-eabi-4.4.0/bin/arm-eabi-addr2line,给您准备了个，咋就用这个吧，功能一样，没啥区别）

     addr2line -e -f libc.so 0001173c  红色是你的目标库，绿色是你出问题的地址，看看#15这行

    结果出现:??pthread\_create

                     ??:0

    恩，好了问题应该出在这个函数中，认为大功告成了？我，呸，还早呢。这个只是告诉你函数入口，至于具体执行到这个函数的哪个地方挂了，还得再看。

c 找到了是哪个部分出了问题，下面就是开启显微镜，看看谁搞的东东。

   使用objdump -S -D libc.so > deassmble\_libc.txt 反汇编下你的动态连接库文件，并且将它写入一个文件中。

   打开这个反汇编过后的重定向文件，在查询的时候输入**1173c**这个偏移地址，你会看到在茫茫人海中

  00011684 <pthread\_create>:  
   11684:       e92d4ff0        push    {r4, r5, r6, r7, r8, r9, sl, fp, lr}  
   11688:       e24dd01c        sub     sp, sp, #28     ; 0x1c  
   1168c:       e1a06001        mov     r6, r1  
   11690:       e1a08002        mov     r8, r2  
   11694:       e1a09003        mov     r9, r3  
   11698:       e3a04001        mov     r4, #1  ; 0x1  
   1169c:       e59f521c        ldr     r5, [pc, #540]  ; 118c0 <pthread\_create+0x23c>  
   116a0:       e58d000c        str     r0, [sp, #12]  
   116a4:       eb009a35        bl      37f80 <strncmp+0x20>  
   116a8:       e59f2214        ldr     r2, [pc, #532]  ; 118c4 <pthread\_create+0x240>  
   116ac:       e1a03000        mov     r3, r0  
   116b0:       e1a01004        mov     r1, r4  
   116b4:       e593c000        ldr     ip, [r3]  
   116b8:       e3a0003c        mov     r0, #60 ; 0x3c  
   116bc:       e08f3005        add     r3, pc, r5  
   116c0:       e7933002        ldr     r3, [r3, r2]  
   116c4:       e5834000        str     r4, [r3]  
   116c8:       e58dc010        str     ip, [sp, #16]  
   116cc:       eb009a3b        bl      37fc0 <strncmp+0x60>  
   ...

**1173c:       ebffec2b        bl      c7f0 <\_\_pthread\_clone>-->就是他了，对你成功了。**

   ...

这个是ARM汇编，需要你翻译成对应的C函数去看，这里我就不做解释了，照着前面的步骤，

对上面中#15-->#00 一共16行慢慢去找，直到找到#00行的问题函数，其实，最后一个#00行的是最重要的（前面不找也行，但是可以多给你提供问题信息，和流程），因为他是 第一目击者，也是Crash前的第一现场。所以找到这个函数很重要，假设我们最后经过万里长针发现#00的出错的地方是pXX->member挂了 (MD，我经常遇到这种问题)。

那么你可以怀疑两个地方：

1 您的指针是空指针，但是现实与理想总是十万八千里，多数情况下很少出现，而且你分析代码后，也会对自己说怎么可能。绝大多数情况下，从我的经验来讲，很少会有空指针这种低级错误，但是不排除哪个2货出现了这么个问题。如果是这个问题，那么恭喜你，你很幸运。

2 还有就是怀疑越界和内存地址被挤占。就拿我以前的经验，指针不是空，但是根据汇编码看，是访问成员变量挂了，这个地址肯定是被占用了。

   针对第2种比较恶心的情况，就需要你看整个log的流程了，需要你看下主要的mainlog关于出现crash前的动作，看看是哪个孙子占用的，以最近原 则为先，从下往上看，唉，说句实在话，李白的一句话可以形容整个过程:"蜀道难，难于上青天啊"。工作量大，而且要细致。我也没什么办法。。。

  还有一种情况，就是内存不够，导致了您的地址被挤占了，出现low memory, no more ...这样的语句，以及大量出现GC\_FOR\_MALLOC关于GC的东西(如果是少量的，可以忽略，大量的话)，呵呵表明你的某个进程在吞噬你的内存， 存在内存泄漏。坑爹啊，这个问题，是最难查的，需要你花大量时间，去看内存的变化。一般看内存的情况是

cat /proc/meminfo

空闲内存=buffer+cache+free这三个字段，Active字段是已经使用的内存，Total不用说，是总的物理内存。(记住 free不高，并不代表你的内存空闲不高，海水不可斗量，需要看全了3个字段的总和才是空闲内存)

如果你想具体跟踪每个进程的内存使用情况，还是在/proc下面，对应了N多的数字文件，那个其实是PID号，进去后cat status可以看到

VMRSS XXXKB就是你当前进程的使用内存量，此外还有一些其他的内存数据，包括页啊等等。。。里面还有很多有用的数据，如果你想跟踪所有的进程的内存情况，推介大家可以看看**linux ps**命令的源码，看看人家是怎么在/proc下遍历进程，并且提取属性值的。

写个daemon，跟踪一段时间，记录下各进程内存的变化，然后就是通宵的分析。。。。

总之，这类问题，很难定位，也很难解决，需要花时间，精力去分析。但是如果你解了，那么相信所有人对你会刮目相看的。包括你的老板，量变引起质变，请记住。

我还在纠结中啊...分析ing

最后再说几句

stack:

#12 476e5eb8  476e5ed8    
    476e5ebc  476e5ed8    
    476e5ec0  00100000    
    476e5ec4  476e5ed8    
    476e5ec8  acaa4d38    
    476e5ecc  005ec9b0    
    476e5ed0  aca4e1a3  /system/lib/libdvm.so  
    476e5ed4  476e5ed8    
#13 476e5ed8  005ec9b0    
    476e5edc  005ecae8    
    476e5ee0  476e5f00    
    476e5ee4  aca4e109  /system/lib/libdvm.so  
    476e5ee8  005ec9b0    
    476e5eec  afd11b98  /system/lib/libc.so  
#14 476e5ef0  476e5f00    
    476e5ef4  005ecae8    
    476e5ef8  45b29b84    
    476e5efc  afd11740  /system/lib/libc.so  
#15 476e5f00  476e5f00    
    476e5f04  005ecae8    
    476e5f08  00000009    
    476e5f0c  00000000    
    476e5f10  00000000    
    476e5f14  005ec9b0    
    476e5f18  00000000    
    476e5f1c  00000000

红色的部分是基地址+偏移地址

比如afd +11740 这个afd你可以从prelink-linux-arm.map找到每个.so的基地址（比如 libc.so                 0xAFD00000 # [~2M]）11740可以从反汇编的文件中找到对应点，但是我不知道这些有什么用。

希望这偏小文章能帮助到大家，大家共同进步，有好的方法也欢迎您一起分享。谢谢！

下面是一个老外的摘录早期存于google论坛的内存分析文章（现在没了），我是在他基础上，又加了一些分析和经验之谈，希望上帝保佑让我早点定位到问题。

How to read Android crash log and stack trace

An Android crash in C/C++ code often generates some crash log which looks like the following. They can be seen either with "adb logcat" or found in one of the tombstones under /data/tombstones. This article briefly explains the structure of the log, how to read it, and how to translate the addresses into symbols with the stack tool.

The basic items in the log is explained below. The first is the build information in the system property "ro.build.fingerprint"

I/DEBUG ( 730): \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\* \*\*\*

I/DEBUG ( 730): Build fingerprint: 'generic/generic/generic/:1.5/...'

Then, it shows the process ID number (pid) and the thread id (tid). Knowing the PID, one can find its information with /proc/<pid> (of course, before the crash happens). In this example, the PID and TID are the same. However, if the crash happens in a child thread, the thread ID tid will be different from pid. The child thread itself is also a process in Linux (having its task\_struct). The difference from a child process created by fork is that it shares address space and other data with the parent process.

I/DEBUG ( 730): pid: 876, tid: 876 >>> /system/bin/mediaserver <<<

The following shows the signal which caused the process to abort, in this case, it's a segmentation fault. This is followed by the register values.

I/DEBUG ( 730): signal 11 (SIGSEGV), fault addr 00000010

I/DEBUG ( 730): r0 00000000 r1 00016618 r2 80248f78 r3 00000000

I/DEBUG ( 730): r4 80248f78 r5 0000d330 r6 80248f78 r7 beaf9974

I/DEBUG ( 730): r8 00000000 r9 00000000 10 00000000 fp 00000000

I/DEBUG ( 730): ip afd020c8 sp beaf98d8 lr 8021edcd pc 8021c630 cpsr a0000030

This is the call stack trace. #00 is the depth of the stack pointer. The "pc <addr>" is the PC address in the stack. Sometimes, the "lr" link register containing the return address is shown instead of PC. It is followed by the file containing the code. Of course, the address doesn't make much sense without resolving it to a symbol. This can be done with the "stack" tool. Download stack here.

I/DEBUG ( 730): #00 pc 0001c630 /system/lib/libhelixplayer.so

I/DEBUG ( 730): #01 pc 0001edca /system/lib/libhelixplayer.so

I/DEBUG ( 730): #02 pc 0001ff0a /system/lib/libhelixplayer.so

I/DEBUG ( 730): #03 pc 000214e0 /system/lib/libutils.so

I/DEBUG ( 730): #04 pc 0000e322 /system/lib/libmediaplayerservice.so

...

I/DEBUG ( 730): #15 pc b0001516 /system/bin/linker

The following is actually the current stack with the stack pointer address and code dump. Each line contains 4 bytes (one machine word), and the address is in ascending order. The words in the stack are mapped onto the memory region it belongs to.

I/DEBUG ( 730): stack:

I/DEBUG ( 730): beaf9898 00016618 [heap]

I/DEBUG ( 730): beaf989c beaf98d0 [stack]

I/DEBUG ( 730): beaf98a0 0000db28 [heap]

I/DEBUG ( 730): beaf98a4 beaf98f8 [stack]

I/DEBUG ( 730): beaf98b8 8021cf4d /system/lib/libhelixplayer.so

I/DEBUG ( 730): beaf98bc 80248f78

I/DEBUG ( 730): #00 beaf98d8 0000d330 [heap]

I/DEBUG ( 730): beaf98dc 00000000

I/DEBUG ( 730): beaf98e0 0000d330 [heap]

I/DEBUG ( 730): beaf98e4 8021edcd /system/lib/libhelixplayer.so

I/DEBUG ( 730): #01 beaf98e8 80248f78

I/DEBUG ( 730): beaf98ec 80248f78