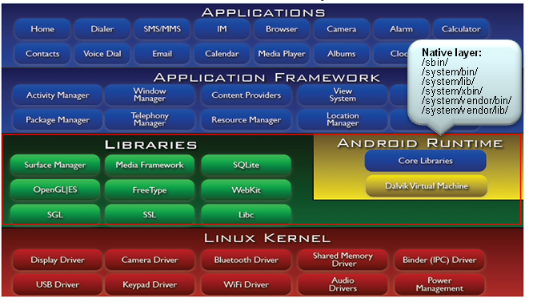
基础篇: 通过log分析NE

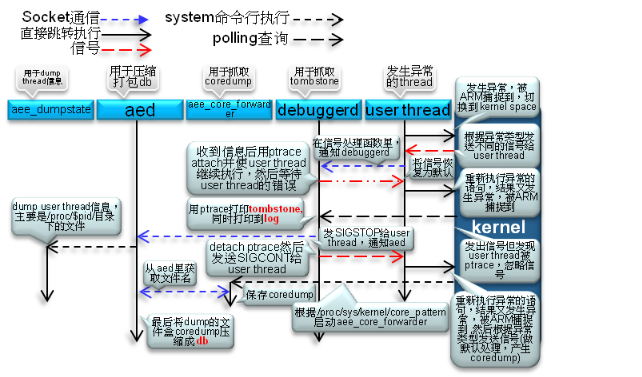
一：NE框架

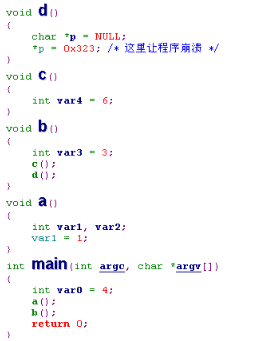
**1. Native application**  
    本地应用程序是指可以直接运行在操作系统上，并且处理器直接执行机器码的程序。  
        比如windows上的各种\*.exe的程序，而linux上的是各种bin程序。  
    在Android上，OS是linux，因此各种bin程序就是所谓natvie application了，比如/system/bin目录下的所有文件。  
    这些应用程序都是由GCC(c/c++)编译生成。  
    在Android软件架构里，这些应用程序组成了native layer:

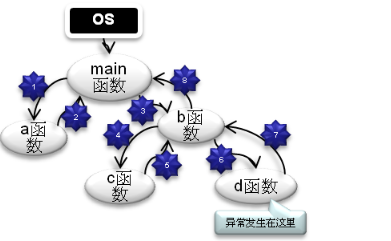


**2. Native Exception**  
    native layer里的应用程序崩溃统称为Native Exception，比如空指针，非法指针，程序跑飞，内存踩坏等，好比像windows下，程序崩溃弹出某某地址不能为read/write。

**3. 总流程图**  
    原始的linux，对于用户进程崩溃之后，处理方式有2种：直接终止进程；输出coredump再终止进程。  
    而在Android，为了方便调试，在收到崩溃信号后，会先输出tombstone，然后在根据设置是否抓取coredump，最后再终止进程。而我司在这之上还会将coredump及其他关键信息打包。  
    以下是完整的NE处理流程图：



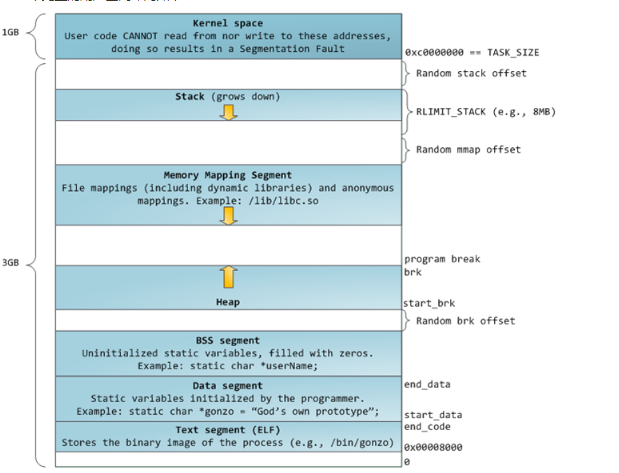
**4. 例子**  
    我们以1个NE的例子来将流程走一遍。  
    首先写test.c：  
  
    然后编写Android.mk：  
LOCAL\_PATH:=$(call my-dir)  
include $(CLEAR\_VARS)  
LOCAL\_SRC\_FILES:=test.c  
LOCAL\_MODULE:=test  
include $(BUILD\_EXECUTABLE)  
    编译后将test推送到手机端，然后执行它，其中走过的函数步骤如下：



 当走到d()函数（也就是第6步之后）时必然产生1个NE。

#### 二：用户空间布局

**1. 布局**  
    native程序是运行在linux之上的，因此程序如何被linux加载起来，如何发生/捕获/处理异常都需要了解的清清楚楚，这样才能系统的分析NE。我们先从user space layout入手。  
    对于32位ARM来说，最大能访问的空间是4G，其中kernel占用了最高1G空间，是所有进程都共享的，剩下的低3G的空间是每个应用程序独有和互不干扰，这样的布局是由MMU帮忙实现的。  
    应用程序内容：  
        主要可以分为3个部分：只读的代码段和数据段，可读写的数据段，bss未初始化数据段。  
    程序的3个部分会被加载到用户空间，同时会分配一个栈给程序，如果程序运行过程中调用malloc()等堆内存分配函数，则会将堆映射到用户空间；如果调用mmap()，则会将文件映射到用户空间。  
    一个完整的用户空间布局如下：



**2. 解剖**  
(1). Text segment  
    也就是程序里的代码段和只读数据段，在ICS及以前版本是固定加载在0x8000的位置上，JB及之后为了安全的考量，则是随机加载在0x40000000的位置上（这个地址之上属于mmap空间）  
    【问题】加载的位置定义在哪里？  
    【解答】在gcc连接脚本里指定，ICS及之前的版本在alps/build/core/armelf.x里有指定\_\_executable\_start = 0x8000，JB及之后的版本则取消该文件了。  
(2). Data segment/BSS segment  
    在Text segment之上就是data和bss了，这3段是由系统帮忙加载，系统是根据应用程序文件里的信息加载的，而应用程序格式在linux上是ELF（如同windows的PE格式）。  
(3). Stack  
    这个栈是系统分配的初始栈，里面存放环境变量，辅助向量，参数等信息。  
    这个栈的大小是受系统限制的，参数是：RLIMIT\_STACK，可以通过ulimit -s查看（单位是KB）：  
        #ulimit -s  
        8192  
    栈底为0xC0000000随机偏移8M（random stack offset）的位置，随机的目的也是为了安全考量。对应内核代码如下：



【问题】如果在程序里创建了其他线程，栈是如何配置的？是否可以自动增长？  
    【解答】线程都是通过pthread库创建，其中pthread\_create()函数就是用于创建线程的，你可以指定栈的空间大小，由该函数帮你申请（通过mmap），这个无法自动增长。  
        也可以自己malloc/mmap等方法申请，再传给该函数，如果是mmap，可以设定是否自动增长。  
(4). Heap  
    应用程序从系统要内存只能通过系统调用：mmap/brk，这里拿到的内存都是按页对齐的（32位ARM是4K）。  
    brk拿到的内存有个特性，向上增长，顶部可以自由伸缩（也就是可以扩展/释放堆）。  
    c常用的malloc首先通过brk调用拿到内存（如何brk失败则通过mmap），然后在经过管理给你指定大小的内存。  
    start\_brk为BSS段随机偏移32M（random brk offset）的位置，随机的目的也是为了安全考量。对应内核代码如下：

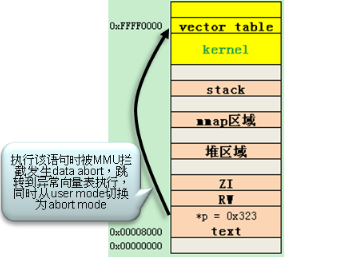


(5). Memory Mapping Segment  
    通过mmap系统调用映射进来的，都会落在这个区域（除非指定了其他地址）。包括动态库和文件。  
    该区域存在2个增长方向：向上增长和向下增长。  
    向上增长：起始地址是TASK\_UNMAPPED\_BASE（0x40000000）。

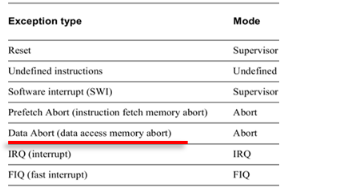
**3. 地址随机化**  
    如上一章节讲到，栈/堆/共享库的位置都有一个随机量，目的是增加攻击者预测目的地址的难度，防止攻击者直接定位攻击代码位置，达到阻止溢出攻击的目的。  
    该技术简称**ASLR**（Address space layout randomization）：一种针对缓冲区溢出的安全保护技术。  
    【问题】如何关闭ASLR？  
    【解答】通过设置kernel.randomize\_va\_space内核参数来设置内存地址随机化的行为：  
        #echo 0 >/proc/sys/kernel/randomize\_va\_space  
        目前randomize\_va\_space的值有三种，分别是[0,1,2]  
            0: 关闭进程地址空间随机化  
            1: 将mmap的基址，stack和vdso页面随机化  
            2: 在1的基础上增加堆(heap)的随机化

#### 三：流程-异常处理

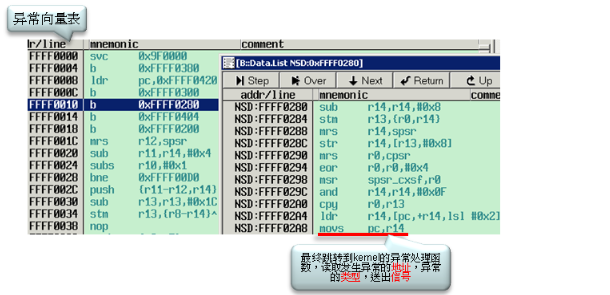
**1. 用户空间产生异常**  
    d()函数里对空指针赋值，由于0地址属于非法地址，访问时会被MMU（内存保护单元）拦截，并发送abort信号给CPU：



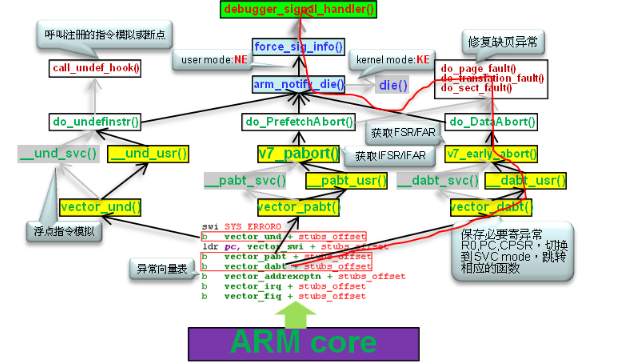
  ARM CPU有1张异常向量表响应对应的异常（可以存放在0x00000000或0xFFFF0000处，目前配置为0xFFFF0000），访问0地址对应的是data abort，如果程序跑飞则对应的是prefetech abort：



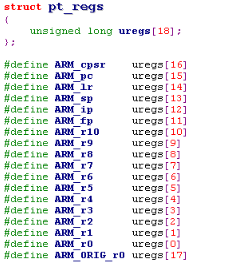
 此时从用户态切换到了内核态，在内核态，会将当前的上下文保存并记录异常的地址和类型，内核处理完成后在返回用户态。  
    以下是对应的异常向量表的汇编代码（对应的代码：alps/kernel/arch/arm/kernel/entry-armv.s）：



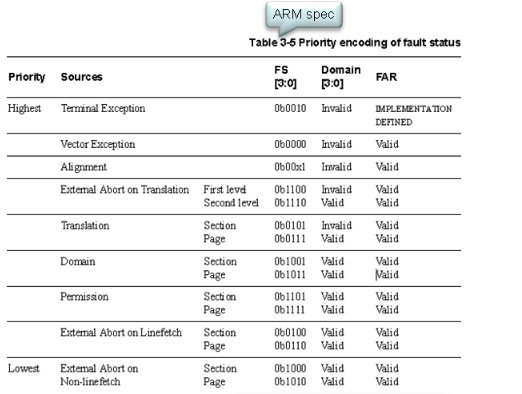
**2. 内核异常处理流程**  
    1张图总结内核异常处理流程：

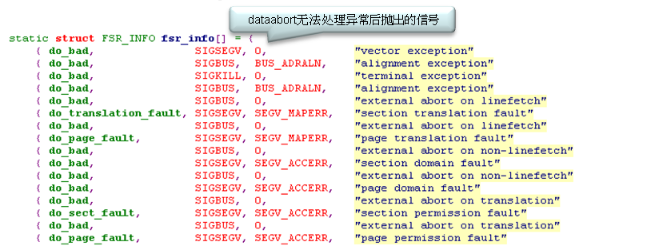


流程按红色的线走。从向量表出发，经过\_dabt\_usr()函数，将上下文保存在该进程的内核栈里（压入pt\_reg），pt\_reg保存了R0~R15,CPSR等信息，处理完后判断是否重新调度，否则弹出pt\_reg返回用户空间：



**3. 内核抛出信号**  
    内核从MMU获取异常类型，然后根据类型调用对应的处理函数：





如果是缺页异常，则会调用do\_page\_fault()/do\_translation\_fault()，最后返回。但是0地址不在进程空间内，因此会转调arm\_notify\_die()，最后发出信号给自己（注意：是自己发出信号给自己！！！）  
    \_\_send\_signal()函数有添加printk：

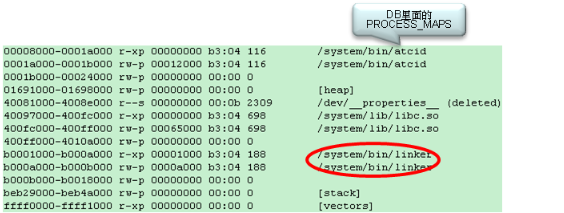


 这时在kernel log看到对应的log：

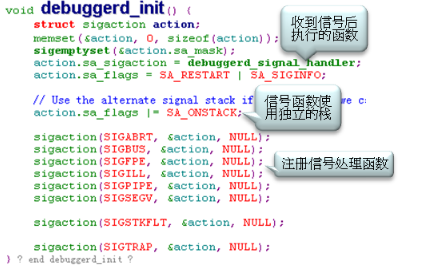


【TIPS】如果发生NE，可以通过kernel log搜索sig 11/7等几个可以导致进程崩溃的关键字判断。  
    进程在收到信号时会做出相应的处理，具体需要查看内核信号章节。  
    到这里，我们在内核绕了1圈又要返回用户空间了。

**4. 信号捕获**  
    大家会想，test.c里没有任何信号处理相关的代码，如何能处理信号？  
    这就需要程序被系统加载说起了。程序分为2种，1种为静态链接的程序，另一种为动态链接的程序。  
    静态链接：由链接器在链接时将库的内容加入到可执行程序中的做法。最大缺点是生成的可执行文件太大，需要更多的系统资源，在装入内存时也会消耗更多的时间。  
    动态链接：将独立的模块先编译成动态库（比如libc.so/libutils.so等），程序运行有需要它们时才加载。最大缺点是可执行程序依赖分别存储的库文件才能正确执行。如果库文件被删除了，移动了，重命名了或者被替换为不兼容的版本了，那么可执行程序就可能工作不正常。  
    在Android里，大部分都是动态链接，而只有init等少部分是静态链接。因此我们的test也是动态链接程序，动态链接程序是需要链接器才能跑起来，linker就是Android的链接器。  
    我们看下/proc/$pid/maps或db里的PROCESS\_MAPS：



inker也在程序的进程空间内。当内核将应用程序加载起来后，并不是先跑应用程序代码，而是先跑linker。linker负责将应用程序所需的库加载到进程空间内，之后才跑应用程序代码。  
    【TIPS】kernel加载完应用程序/动态链接器后并不帮忙加载所需的动态库，这些工作只能由动态链接器完成。动态链接器可以使用2种方式加载动态库：立即模式：将所需的动态库一次性全部加载，这样可以提高性能但延长加载时间。懒惰模式：程序需要的时候再加载，前期加载时间快但是如果程序在执行关键的代码，动态链接会影响性能。  
    linker执行期间还做了一件事：注册信号！，具体的函数调用流程如下：  
        \_\_linker\_init() -> \_\_linker\_init\_post\_relocation() -> debuggerd\_init()



信号的知识，请查看：编程篇: linux c编程的信号章节。

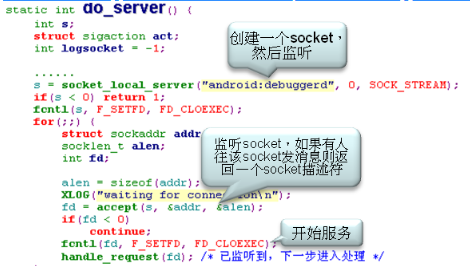
5. 信号处理  
    目前会产生native exception（NE）的几个信号需要特别掌握产生的原因，这样才能进一步分析问题所在。  
    内核发送信号过来后会执行debuggerd\_init()里注册的函数debugger\_signal\_handler()，该函数会打印基本信息到main log：  
        logSignalSummary()函数会输出基本异常信息：



然后连接debuggerd的socket，将pid等信息发送给debuggerd，请它帮忙后续的处理。之后将对应的信号恢复为默认，等待debuggerd处理完后抓取coredump。

#### 四：流程-debuggerd

**1. debuggerd服务**  
    init.rc会将debuggerd拉起来，具体代码请查看system/core/debuggerd/目录。debuggerd起来后会创建1个socket，然后监听，等待别人通过socket请求服务，服务可以是生成tombstone或调用栈：



【TIPS】当然你也可以单独调用debuggerd  
        debuggerd -b $tid。抓取指定tid的调用栈。  
        debuggerd $tid。抓取指定tid的tombstone。

**2. ptrace attach/detach**  
    通过socket拿到tid等资料后，使用ptrace attach上目标tid，之后就可以通过ptrace访问目标进程空间，然后打印一些NE相关的寄存器/调用栈等信息。  
(1). ptrace attach  
    ptrace attach会发送sig 19给对应的进程。在这里，我们将进程内所有线程都attach上，防止有线程提前退出。从log可以看到这个动作：



(2). ptrace cont  
    attach之后还不能直接访问目标进程，因为目标进程还处于信号处理函数里面，我们需要让它恢复到异常现场，因此需要用ptrace cont让其继续执行。  
(3). waitpid  
    程序接着往下跑必然会再次发生异常（如果是SIGABRT,SIGFPE,SIGPIPE,SIGSTKFLT，则会在信号处理函数补刀，重发一次信号），kernel会再次发出信号，只不过由于进程被ptrace了，信号会发送给debuggerd。

    【TIPS】到这里，目标进程已经收到2次同样的信号了！！！  
(4). tombstone  
    debuggerd收到信号后，就可以生成tombstone了。  
(5). ptrace detach  
    完成工作后，需要detach ptrace，然后发送sig 18让其继续奔跑。如果是访问空指针等错误，程序会再次发生异常，由于在信号处理函数里已经将对应信号恢复默认，因此可能会产生coredump。  
    详细代码分析如下（省略无关紧要部分）：

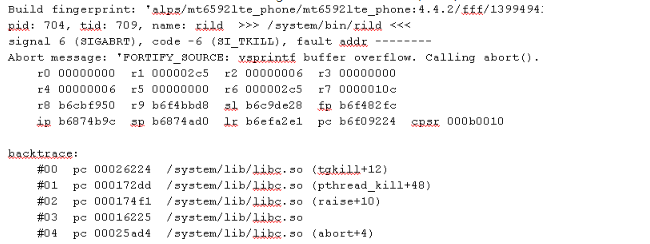


**3. tombstone**  
(1). 创建1个tombstone文件。  
    最多10个，如果已存在10个，则覆盖最旧的文件。  
(2). 版本信息  
    主要是fingerprint，可以看出异常版本是eng还是user。  
(3). 寄存器信息  
    主要查看是哪个进程崩溃，信号是什么。寄存器信息需要配合下面的调用栈信息及数据信息结合GNU的工具（objdump -S反汇编）分析。  
(4). 调用栈信息  
    这个是最直接可以看出异常的信息。  
(5). 其他线程信息  
    如果异常线程和其他线程有逻辑关系的话，可以查看对应线程的信息。  
(6). main log信息  
    只会印部分log信息，全面的log建议还是查看main log。  
    详细代码分析如下（省略无关紧要部分）：



4. 例子解读

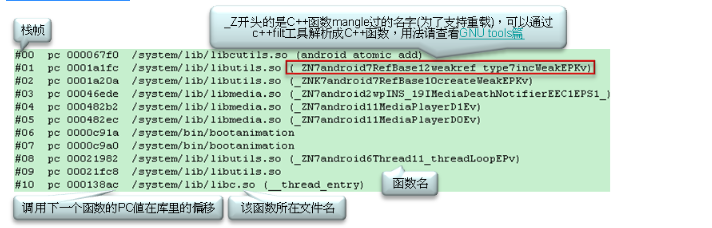
    至此tombstone就完全输出到文件或main log里了，大致的格式如下（例子）：



可以看到NE的原因是sig 6，有人调用了aboart（类似于assert），原因也印出来了，在abort message里。这就需要从调用栈入手分析（当然这个例子看不到是谁调用了abort()）。

#### 五：流程-调用栈

概要  
    上一章节讲到debuggerd打印调用栈，调用栈是分析问题的关键，因此这边单独拿出来讲解。  
    目前的C/C++语言的过程调用都需要栈，正在执行的函数有属于自己的栈帧，函数内部的局部变量就放在栈帧里，当然还会存放函数的返回地址，这样函数执行结束之后才知道返回到哪里。  
    不同的栈帧关联在一起就会形成一个调用链，最顶端表示当前正在执行的函数，第2行表示调用它的函数，以此类推。  
    下面将讲解Andoird上调用栈的形成和解析。  
1. 解读  
    不同版本的Android的debuggerd打印的调用栈有些差别。  
    ICS及以前的版本：



JB及以后的版本（差别在于函数名做了demangle，更容易看懂）：



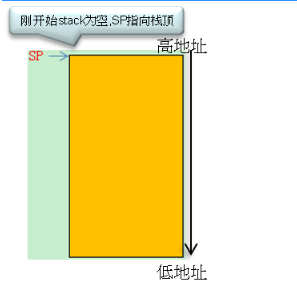
   还原有时可能出现问题，只有2层：



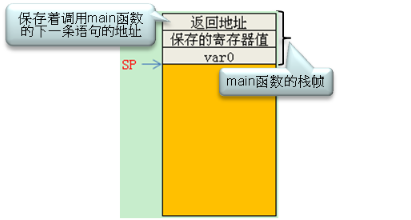
【TIPS】有时不需要拿到含有debug信息的库/应用程序而仅通过函数名就可以分析问题了。

2. 栈帧  
    调用栈的原因只有了解清楚了，才能更好的分析问题。这边会讲解随着代码的执行，栈帧的构造过程。下面以test程序为例：

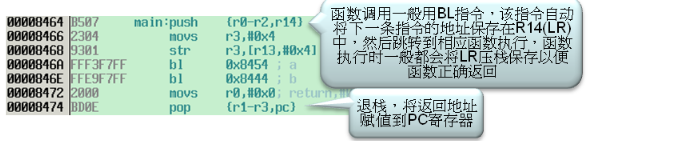
(-1). 程序执行前  
    内核会分配默认的栈给进程的第1个线程使用，此时栈是空的（ARM的栈是满递减类型），SP在栈顶：



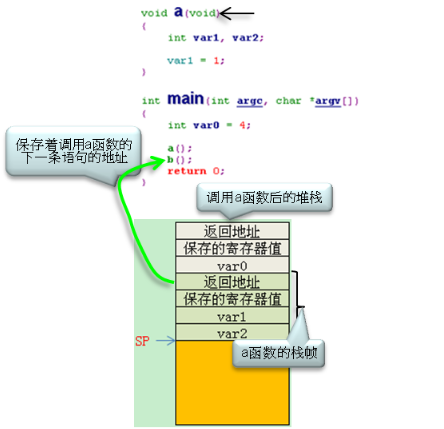
(0). 执行main()函数  
    根据main()函数的汇编代码，前面做了压栈和保存一些必要的寄存器的动作，之后SP向下(低地址，栈是从上往下增长)移动，栈布局如下：



对应的汇编指令如下：

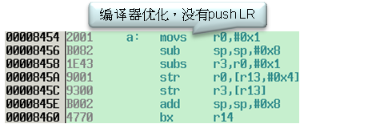


【问题】谁调用了main()函数，是OS吗？  
    【解答】linker跑完后，会调用程序的\_start()而非main()函数，原因是链接脚本定义了入口函数是\_start()，而且要在main()函数前做很多准备工作，大家可以去看bionic/libc/arch-arm/bionic/crtbegin.c。  
(1). 执行a()函数  
    此时会在main()函数之下再压入a()函数的栈帧：

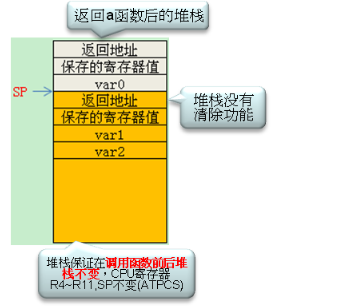


对于一个栈帧来说，布局不一定都是一样的，比如有的可能没有保留返回地址，有的没有保存寄存器，编译器会根据情况做到最优化。

    比如以下函数就没有保留返回地址，那返回地址记录在哪里呢？ARM有很多寄存器，有时用不了这么多，因此就保留在寄存器上了，可以减少对内存的访问，提升性能：



(2). 退出a()函数  
    这时a()函数的栈帧就要被弹出了，SP将往上移动。至于之前属于a()的栈帧一般还保留着，只不过调用下一个函数后，将被覆盖掉。

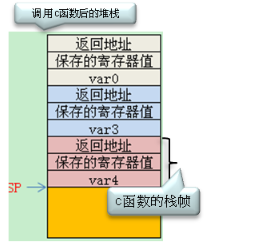


【TIPS】局部变量的生命周期就是进入函数到离开函数这段时间。

(3). 调用b()函数  
    原先a()函数的栈帧就被覆盖了（从栈里看不出之前执行过的函数的痕迹），SP向下移动。

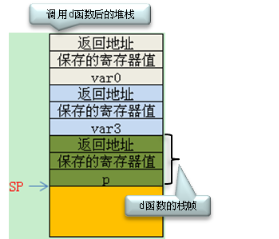


(4). 调用c()函数  
    在b()函数之下有压入c()函数的栈帧，此时有3个栈帧了。



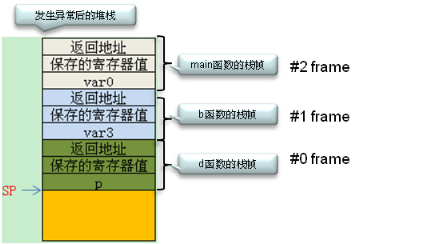
(5). 退出c()函数  
    这里和退出a()函数差不多，这边就省略了。

(6). 调用d()函数  
    这里和调用b()函数差不多。

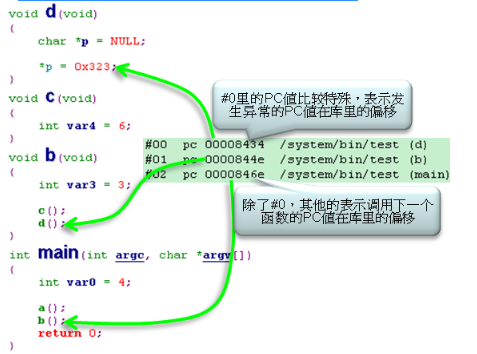


3. NE现场

    在d()函数里，由于对空指针赋值会引发SIGSEGV，所以最后栈的样子如下：



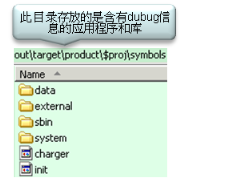
 从tombstone输出的调用栈的栈帧里的地址有特别的含义：



【TIPS】库里的偏移：库默认都加载在0地址的，由OS随机加载在mmap区域，因此实际上库函数的地址都是基址+偏移量，其中的基址就是库加载的地址，这个地址可以从/proc/$pid/maps看到每个库的基址。

    这些偏移地址该如何使用呢？

    其实我们更想知道该函数所在的源文件和所在行数，而透过PC值就可以给出这样的结果，那是是谁将PC值和源文件/行数关联起来呢？  
    实际上Android已经做好这一切了，编译每一个库和程序，都会为其生成一个带有调试信息的库或程序。这个关联就存放在含有调试信息的库/程序中，可以通过addr2line这个工具将PC值转化为源文件和行数。



【注意】含有调试信息的库或程序必须和手机里的库或程序是同一次编译生成的，否则得到的PC值和库里的调试信息不匹配。

    【问题】如何确定库或程序含有调试信息呢？

    【解答】用file命令即可查看：file xxx

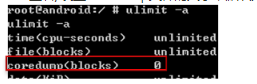
        如果没有含debug信息的话，会显示：  
        xxx: ELF 32-bit LSB shared object, ARM, version 1 (SYSV), dynamically linked, stripped

        含有调试信息的话，则显示：  
        xxx: ELF 32-bit LSB shared object, ARM, version 1 (SYSV), dynamically linked, not stripped

    【TIPS】addr2line的用法请查看GNU tools章节。

#### 六：流程-产生db

**概要**  
    当debuggerd处理完后，程序再次回到发生异常的环境，此时还会发生1次异常，同样的会再次发送信号。  
    由于linker里的debuggerd\_init()里注册的几个信号，默认行为都会产生coredump，因此接下来会介绍coredump产生以及db打包过程。  
    【TIPS】到这里，目标进程已经收到3次同样的信号了！！！  
**1. 产生coredump**  
    当一个程序崩溃时，OS会将该进程的的地址空间保存起来，然后通过工具(GDB,trace32)离线调试。  
        有的问题仅仅通过backtrace是无法直接定位问题的 (指针错误，访问无效内存，内存被踩坏，函数参数错误)。  
    coredump默认是关闭的，并且有些参数可以设置它，如下：  
(1). 参数  
    /proc/sys/kernel/core\_pattern  
        这个参数用于设置coredump文件的名称，支持的参数有  
            %p: 添加pid                                   %u: 添加当前uid  
            %g: 添加当前gid                             %s: 添加导致产生core的信号  
            %t: 添加core文件生成时的unix时间  %h: 添加主机名  
            %e: 添加命令名  
        在aed起来时会对其做初始化  
            eng build: |/system/bin/aee\_core\_forwarder /data/core/ %p %s UID=%u GID=%g  
            user build: /bad\_core\_pattern  
    ulimit -a  
        查看/设置coredump文件的大小，默认为0，也就是不抓coredump



 可以用ulimit -c <filesize>(KB)改变大小  
        【TIPS】当core\_pattern里有管道时忽略此参数(也就是第1个字符为'|')！  
    /proc/$pid/coredump\_filter  
        coredump是抓取进程空间内的内存并保存到文件上，并不是所有内存都需要保存的，你可以通过该参数过滤，只抓取部分内存。  
        该参数是一个值，每个bit位都有对应的含义，用来表示是否抓取这部分内存。  
            bit0: 私有匿名                      bit1: 共享匿名  
            bit2: 有底层文件的私有映射   bit3: 有底层文件共享映射  
            bit4: ELF头                          bit5: 私有大尺寸页  
            bit6: 共享大尺寸页  
        当前默认值是0x23，也就是只会抓取：私有匿名/共享匿名/私有大尺寸页  
(2). 抓取  
    在eng版本，/proc/sys/kernel/core\_pattern被设置为|/system/bin/aee\_core\_forwarder /data/core/ %p %s UID=%u GID=%g，也就是说进程内存数据会导向aee\_core\_forwarder这个程序。  
    aee\_core\_forwarder会向aee询问要保存到哪里，aee会提供db所在路径，然后倒入该路径，最后由aee统一压缩为db。



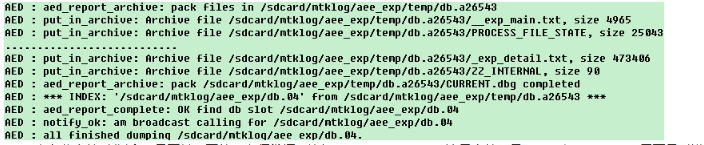
【TIPS】当coredump没有正常生成时可以通过log分析问题点  
    如果aee没有反馈，则aee\_core\_forwarder会保存到/data/core/目录下以zcore-xxx.zip文件（可以用GAT解压）保存。当然了data空间很宝贵，如果生成coredump后如果空间小于4M，则删除coredump。



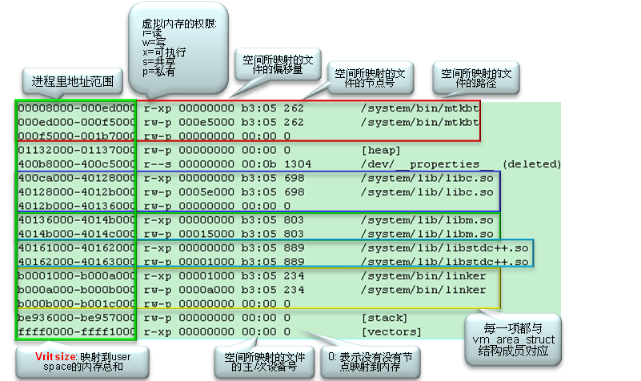
(3). 无coredump原因和解决方法  
    有时候发现进程崩溃了，但是没有db或有db但是没有coredump。这边总结如下：  
        1. 进程重新注册了几个NE的信号，导致异常时被捕获了，没有正常的NE流程。  
        2. init之中发生NE。init进程如果有人尝试杀死则会演变为KE。  
        3. 存储空间满。  
        4. 进程异常后其他线程提前退出。  
        5. fd leak，导致无法打开socket，只能直接触发coredump放在/data/core目录下。  
        6. user版本没有开mtkloger也是没有coredump的。  
        ......  
    解决方法  
        adb shell aee -d coreon之后删除/system/bin/debuggerd。然后重启机器设置adb shell echo "|/system/bin/aee\_core\_forwarder /data/core/ %p %s UID=%u GID=%g" > /proc/sys/kernel/core\_pattern（注意重启后失效）  
        复现问题，coredump可以在data/core目录下找到。  
(4). 主动触发coredump  
    有时为了分析问题，需要主动触发coredump，然后通过工具分析。  
    你可以通过adb shell kill -5 $pid或连发3次adb shell kill -11 $pid。  
    coredump会在/data/core或db里面。  
**2. 产生db**  
    debuggerd完成之后会通知aee，aee就开始了打包db的工作。  
    1个完整的NE的db，里面除了coredump还有其他文件，这些文件绝大部分是通过aee\_dumpstate保存起来的。



整个压缩过程会有log印出来，因此如果有什么问题也可以通过log分析。



db中有些文件对分析NE是至关重要的，必须掌握。比如PROCESS\_MAPS，这只文件就是/proc/$pid/maps，里面是对进程空间的描述：



1). db解包  
    db需要GAT工具才可以解包，更多细节请查看DCC上的GAT使用文档。

    大致结果如下：



(2). db相关配置  
    db个数：user版本：4个，eng版本：20个  
    coredump只有在以下设置才会抓取：  
        eng版本  
        user版本+打开mtklogger  
    详情请看DCC上的文档：MediaTek Logging SOP.pptx

结语

    至此，整个NE的流程就结束了，那么大家就应该可以简单分析NE产生的问题了。coredump分析将会在下一章节讲解。

#### 七：流程-debuggerd on android P

1. tombstoned服务  
P 版本上移除掉了debuggerd 进程，多了一个tombstoned 进程

在系统启动过程中，会将tombstoned trigger 起来。 tombstoned 会创建socket，然后监听.

等待别人通过socket来请求服务。tombstoned返回backtrace 文件或者tombstoned 文件的文件描述符。

具体code 是在system/core/debuggerd/tombstoned/tombstoned.cpp

int [main](http://10.16.20.15/alps-mp-p0-mp3/s?refs=main&project=system)(int, char\* []) {

[398](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "398)   [umask](http://10.16.20.15/alps-mp-p0-mp3/s?defs=umask&project=system)(0137);

[399](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "399)

[400](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "400)   // Don't try to connect to ourselves if we crash.

[401](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "401)   struct [sigaction](http://10.16.20.15/alps-mp-p0-mp3/s?defs=sigaction&project=system) [action](http://10.16.20.15/alps-mp-p0-mp3/s?refs=action&project=system) = {};

[402](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "402)   [action](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#action).[sa\_handler](http://10.16.20.15/alps-mp-p0-mp3/s?defs=sa_handler&project=system) = [](int [signal](http://10.16.20.15/alps-mp-p0-mp3/s?refs=signal&project=system)) {

[403](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "403)   [LOG](http://10.16.20.15/alps-mp-p0-mp3/s?defs=LOG&project=system)([ERROR](http://10.16.20.15/alps-mp-p0-mp3/s?defs=ERROR&project=system)) << "received fatal signal " << [signal](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#signal);

[404](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "404)   [\_exit](http://10.16.20.15/alps-mp-p0-mp3/s?defs=_exit&project=system)(1);

[405](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "405)   };

[406](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "406)   [debuggerd\_register\_handlers](http://10.16.20.15/alps-mp-p0-mp3/s?defs=debuggerd_register_handlers&project=system)(&[action](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#action));

[407](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "407)

[408](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "408)   int [intercept\_socket](http://10.16.20.15/alps-mp-p0-mp3/s?refs=intercept_socket&project=system) = [android\_get\_control\_socket](http://10.16.20.15/alps-mp-p0-mp3/s?defs=android_get_control_socket&project=system)([kTombstonedInterceptSocketName](http://10.16.20.15/alps-mp-p0-mp3/s?defs=kTombstonedInterceptSocketName&project=system));

[409](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "409)   int [crash\_socket](http://10.16.20.15/alps-mp-p0-mp3/s?refs=crash_socket&project=system) = [android\_get\_control\_socket](http://10.16.20.15/alps-mp-p0-mp3/s?defs=android_get_control_socket&project=system)([kTombstonedCrashSocketName](http://10.16.20.15/alps-mp-p0-mp3/s?defs=kTombstonedCrashSocketName&project=system));

[410](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "410)

[411](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "411)   if ([intercept\_socket](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#intercept_socket) == -1 || [crash\_socket](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#crash_socket) == -1) {

[412](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "412)   [PLOG](http://10.16.20.15/alps-mp-p0-mp3/s?defs=PLOG&project=system)([FATAL](http://10.16.20.15/alps-mp-p0-mp3/s?defs=FATAL&project=system)) << "failed to get socket from init";

[413](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "413)   }

[414](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "414)

[415](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "415)   [evutil\_make\_socket\_nonblocking](http://10.16.20.15/alps-mp-p0-mp3/s?defs=evutil_make_socket_nonblocking&project=system)([intercept\_socket](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#intercept_socket));

[416](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "416)   [evutil\_make\_socket\_nonblocking](http://10.16.20.15/alps-mp-p0-mp3/s?defs=evutil_make_socket_nonblocking&project=system)([crash\_socket](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#crash_socket));

[417](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "417)

[418](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "418)   [event\_base](http://10.16.20.15/alps-mp-p0-mp3/s?defs=event_base&project=system)\* [base](http://10.16.20.15/alps-mp-p0-mp3/s?refs=base&project=system) = [event\_base\_new](http://10.16.20.15/alps-mp-p0-mp3/s?defs=event_base_new&project=system)();

[419](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "419)   if (![base](http://10.16.20.15/alps-mp-p0-mp3/s?defs=base&project=system)) {

[420](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "420)   [LOG](http://10.16.20.15/alps-mp-p0-mp3/s?defs=LOG&project=system)([FATAL](http://10.16.20.15/alps-mp-p0-mp3/s?defs=FATAL&project=system)) << "failed to create event\_base";

[421](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "421)   }

[422](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "422)

[423](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "423)   [intercept\_manager](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#intercept_manager) = new [InterceptManager](http://10.16.20.15/alps-mp-p0-mp3/s?defs=InterceptManager&project=system)([base](http://10.16.20.15/alps-mp-p0-mp3/s?defs=base&project=system), [intercept\_socket](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#intercept_socket));

[424](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "424)

[425](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "425)   [evconnlistener](http://10.16.20.15/alps-mp-p0-mp3/s?defs=evconnlistener&project=system)\* [tombstone\_listener](http://10.16.20.15/alps-mp-p0-mp3/s?refs=tombstone_listener&project=system) =

[426](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "426)   [evconnlistener\_new](http://10.16.20.15/alps-mp-p0-mp3/s?defs=evconnlistener_new&project=system)([base](http://10.16.20.15/alps-mp-p0-mp3/s?defs=base&project=system), [crash\_accept\_cb](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#crash_accept_cb), [CrashQueue](http://10.16.20.15/alps-mp-p0-mp3/s?defs=CrashQueue&project=system)::[for\_tombstones](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#for_tombstones)(), [LEV\_OPT\_CLOSE\_ON\_FREE](http://10.16.20.15/alps-mp-p0-mp3/s?defs=LEV_OPT_CLOSE_ON_FREE&project=system),

[427](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "427)   -1 /\* backlog \*/, [crash\_socket](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#crash_socket));

[428](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "428)   if (![tombstone\_listener](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#tombstone_listener)) {

[429](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "429)   [LOG](http://10.16.20.15/alps-mp-p0-mp3/s?defs=LOG&project=system)([FATAL](http://10.16.20.15/alps-mp-p0-mp3/s?defs=FATAL&project=system)) << "failed to create evconnlistener for tombstones.";

[430](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "430)   }

[431](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "431)

[432](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "432)   if ([kJavaTraceDumpsEnabled](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#kJavaTraceDumpsEnabled)) {

[433](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "433)   const int [java\_trace\_socket](http://10.16.20.15/alps-mp-p0-mp3/s?refs=java_trace_socket&project=system) = [android\_get\_control\_socket](http://10.16.20.15/alps-mp-p0-mp3/s?defs=android_get_control_socket&project=system)([kTombstonedJavaTraceSocketName](http://10.16.20.15/alps-mp-p0-mp3/s?defs=kTombstonedJavaTraceSocketName&project=system));

[434](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "434)   if ([java\_trace\_socket](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#java_trace_socket) == -1) {

[435](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "435)   [PLOG](http://10.16.20.15/alps-mp-p0-mp3/s?defs=PLOG&project=system)([FATAL](http://10.16.20.15/alps-mp-p0-mp3/s?defs=FATAL&project=system)) << "failed to get socket from init";

[436](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "436)   }

[437](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "437)

[438](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "438)   [evutil\_make\_socket\_nonblocking](http://10.16.20.15/alps-mp-p0-mp3/s?defs=evutil_make_socket_nonblocking&project=system)([java\_trace\_socket](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#java_trace_socket));

[439](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "439)   [evconnlistener](http://10.16.20.15/alps-mp-p0-mp3/s?defs=evconnlistener&project=system)\* [java\_trace\_listener](http://10.16.20.15/alps-mp-p0-mp3/s?refs=java_trace_listener&project=system) =

[440](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "440)   [evconnlistener\_new](http://10.16.20.15/alps-mp-p0-mp3/s?defs=evconnlistener_new&project=system)([base](http://10.16.20.15/alps-mp-p0-mp3/s?defs=base&project=system), [crash\_accept\_cb](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#crash_accept_cb), [CrashQueue](http://10.16.20.15/alps-mp-p0-mp3/s?defs=CrashQueue&project=system)::[for\_anrs](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#for_anrs)(), [LEV\_OPT\_CLOSE\_ON\_FREE](http://10.16.20.15/alps-mp-p0-mp3/s?defs=LEV_OPT_CLOSE_ON_FREE&project=system),

[441](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "441)   -1 /\* backlog \*/, [java\_trace\_socket](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#java_trace_socket));

[442](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "442)   if (![java\_trace\_listener](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp#java_trace_listener)) {

[443](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "443)   [LOG](http://10.16.20.15/alps-mp-p0-mp3/s?defs=LOG&project=system)([FATAL](http://10.16.20.15/alps-mp-p0-mp3/s?defs=FATAL&project=system)) << "failed to create evconnlistener for java traces.";

[444](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "444)   }

[445](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "445)   }

[446](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "446)

[447](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "447)   [LOG](http://10.16.20.15/alps-mp-p0-mp3/s?defs=LOG&project=system)([INFO](http://10.16.20.15/alps-mp-p0-mp3/s?defs=INFO&project=system)) << "tombstoned successfully initialized";

[448](http://10.16.20.15/alps-mp-p0-mp3/xref/system/core/debuggerd/tombstoned/tombstoned.cpp" \l "448)   [event\_base\_dispatch](http://10.16.20.15/alps-mp-p0-mp3/s?defs=event_base_dispatch&project=system)([base](http://10.16.20.15/alps-mp-p0-mp3/s?defs=base&project=system));  
  
  
2. debuggerd\_signal\_handler 处理流程  
  
进程启动过程中，依然是在linker 来call 到debuggerd\_init(), 注册signal handle。 这和之前版本一样。  
当进程收到signal ，call 到signal 处理函数，debugger\_signal\_handler 处理流程大致如下:

进阶篇: coredump分析

#### GNU tools

1. 概要  
       Android编译工具使用了gcc，因此了解和使用gnu toolchain可以在debug时更加方便。  
       工具目录(这里只列了ARM平台相关的工具)  
           ARM32位版本：prebuilts/linux-x86/gcc/arm/arm-linux-androideabi-$version/bin
2. ARM64位版本：prebuilts/linux-x86/gcc/aarch64/aarch64-linux-android-$version/bin  
       为了方便使用，可以将目录添加到环境变量中  
           编辑$HOME/.bash\_profile文件，在最后添加以下语句即可将$HOME/bin加入环境变量,$HOME/bin也可以改为你想添加的路径：  
               PATH=$PATH:$HOME/bin  
               export PATH  
   1. addr2line  
       将地址转换为地址所在的文件及行数(显示所在函数)  
       使用方法：arm-linux-androideabi-addr2line [option(s)] [addr(s)]  
   (1). 参数 (常用)  
       -e --exe=<executable>: 设置要查询地址的文件(默认: a.out)  
           一般是\*.so/\*.a和可执行程序  
           此文件必须带有debug信息，在android codebase里是放在out/target/product/$project/symbols目录下  
       -f –functions: 显示地址所在的函数名  
       -C --demangle[=style]: 反重整函数名为可读方式  
           自动识别格式，C++函数才需要此参数  
   (2). 例子  
       arm-linux-androideabi-addr2line -e libc.so -f -C 0x23234  
       wcscoll  
       alps/bionic/libc/wchar/wcscoll.c:37  
   (3). 什么情况下需要用到？  
       发生NE后，会生成tombstones/tombstones\_xx文件或aee\_exp里的db解开之后的\_\_exp\_main.txt ，里面有backtrace信息，就可以通过addr2line分析出哪个文件哪行哪个函数（注意用-e 载入的文件必须和手机的bin档同一次编译生成，否则地址和符号可能不一致）  
   2. nm  
       arm-linux-androideabi-nm [option(s)] [file(s)]  
       列出该文件的符号(函数，变量，文件等)，包含名字、地址、大小  
   (1). 参数 (常用)  
       -C, --demangle[=STYLE]: 反重整符号为可读方式  
           自动识别格式  
       -e --exe=<executable>: 设置要查询地址的文件(默认: a.out)  
           一般是\*.so和可执行程序  
       -D, --dynamic: 只显示动态符号  
       -g, --extern-only: 只显示外部符号  
       -l, --line-numbers:多显示符号所在文件和行数  
       -S, --print-size: 多显示符号的大小  
       -u, --undefined-only: 只显示未定义的符号  
   (2). 符号类型 (常用)  
       小写表示是本地符号，大写表示全局符号(external)  
       A: 符号值是绝对的。在进一步的连接中，不会被改变(absolute)  
       B: 符号位于未初始化数据段(BSS section)  
       C: 共用(common)符号. 共用符号是未初始化的数据。在连接时，多个共用符号可能采用一个同样的名字，如果这个符号在某个地方被定义，共用符号被认为是未定义的引用  
       D: 已初始化数据段的符号(data section)  
       F: 源文件名称符号  
       R: 只读数据段符号.(定义为const的变量)  
       T: 代码段的符号 (text section)  
       U: 未定义符号  
       ?: 未知符号类型，或者目标文件特有的符号类型  
   (3). 例子  
       arm-linux-androideabi-nm -g test  
           00009018 D \_\_CTOR\_LIST\_\_  
           00009010 T \_\_FINI\_ARRAY\_\_  
           00009008 T \_\_INIT\_ARRAY\_\_  
                    U \_\_aeabi\_unwind\_app\_pr0  
           ......  
   3. objdump  
       arm-linux-androideabi-objdump <option(s)> <file(s)>  
       查看对象文件(\*.so/\*.a或应用程序)的内容信息  
   (1). 参数 (常用)  
       至少需要一个以下的参数  
           -a, --archive-headers: 显示库(\*.a)成员信息  
           -f, --file-headers:显示obj中每个文件的整体头部摘要信息  
           -h, --[section-]headers:显示目标文件各个section的头部摘要信息  
           -x, --all-headers: 显示所有头部摘要信息  
           -d, --disassemble:反汇编代码段  
           -D, --disassemble-all: 反汇编所有段  
           -S, --source:反汇编出源代码，额外有debug信息，隐含-d，如果编译时有-g，效果更明显  
           -t, --syms: 显示符号表  
           -r, --reloc: 显示重定位记录  
       -C, --demangle[=STYLE]: 反重整符号为可读方式  
           自动识别格式  
   (2). 例子  
       arm-linux-androideabi-objdump -S libstdc++.so > disas.txt  
   (3). 什么情况下需要用到？  
       当发生NE后，拿到backtrace可以查看对应地址的汇编代码。  
   4. readelf  
       arm-linux-androideabi-readelf <option(s)> elf-file(s)  
       查看elf文件(\*.so/\*.a或应用程序)的内容信息  
   (1). 参数 (常用)  
       -a, --all: 显示所有可显示的内容  
       -h --file-header: 显示ELF文件头  
       -l --segments: 显示程序头组  
       -S --sections: 显示节头组  
       -t: 显示节头细节  
       -e --headers: 等效于-h -l -S  
       -s --syms: 显示符号表  
       -n --notes: 显示内核说明  
       -r --relocs: 显示重定位信息  
       -u --unwind: 显示解栈信息  
       -d --dynamic: 显示动态节  
       -p –string-dump=<num|name>: 以字符串的方式显示节  
       -W --wide: 允许一行显示超过80个字符  
   (2). 例子  
       arm-linux-androideabi-readelf -a -W adb > 1.txt  
   (3). 什么情况下需要用到？  
       学习/查看ELF结构。  
   5. c++filt  
       arm-linux-androideabi-c++filt <function name>  
       反重整C++符号为可读方式  
   (1). 例子  
       arm-linux-androideabi-c++filt \_ZN20android\_audio\_legacy22AudioPolicyManagerBase17setSystemPropertyEPKcS2  
       android\_audio\_legacy::AudioPolicyManagerBase::setSystemProperty(char const\*, char const\*)  
   (2). 什么情况下需要用到？  
       通过GAT解开db后(输入symbols目录)，如果是NE的话，可以查看解开的C++的对应的symbols是否和c++filt的一样，不一样则symbols目录和手机bin档不匹配。

#### 2．AAPCS标准

**概要**  
    使用工具分析coredump，不可避免要接触ARM汇编。汇编里的指令的含义可以查看spec了解，而函数参数的传递方法，栈的布局，寄存器的规划是查看汇编的重要部分，这部分由ARM AAPCS定义。熟悉它才能理解和分析coredump。

**1. Procedure Call Standard for the ARM Architecture**  
    The layout of data  
    Layout of the stack and calling between functions with public interfaces  
    Variations available for processor extensions, or when the execution environment restricts the addressing model  
    The C and C++ language bindings for plain data types  
    熟悉AAPCS后，查看汇编代码将有很大帮助。

**2. 文档来源**  
    ARM官网<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/index.html>  
    ARM软件开发工具 => ABI for the ARM Architecture => Procedure Call Standard for the ARM Architecture

#### **3. Core registers**

#### R0-R3(a1-a4)         子程序间通过r0-r3来传递子程序前4个参数(剩余参数通过压栈传递，8个字节的double类型由两个寄存器一起传递)         子程序返回结果，也是使用r0-r3来传递，如果组合类型(编译时不确定长度的)超过4字节的将结果存放在memory，然后返回地址     R4-R8, R10 and R11 (v1-v5, v7 and v8)         用于保存局部变量(thumb用R4-R7但不局限于R4-R7)     R9         平台相关，比如：             在数据位置无关模型里作为：static base(SB)             作为TLS指针：thread register(TR)             或是作为普通的v6     R12         r12用作子程序间(子程序如果太远需要胶合代码辅助跳转)scratch寄存器，别名为ip         也可以用来保存中间值     R13-R15         特殊用途，分别为SP(栈指针),LR(保存子程序的返回地址),PC(程序计数器)         LR保存后，可以客串下保存中间值     在调用子程序前后，保证R4-R8, R10, R11和SP(如果R9作为v6，则包括R9)的值不变         如果在子程序内有使用以上寄存器，那么必须在子程序头做压栈保存寄存器值，在退出之程序前做出栈回复寄存器值。

#### **4. 堆栈/子程序调用**     堆栈为FD类型，对堆栈的操作是8字节对齐的。使用stmdb/ldmia批量内存访问指令来操作FD堆栈         LDRD/STRD要求数据栈是8字节对齐的,以提高数据的传送速度         比如：STMFD sp!,{R4-R11,LR,PC}, PC是不需要压栈保存的，这里压栈仅仅是为了保证8字节边界对齐     子程序调用         需要完成以下动作             LR[31:1]保存返回地址，LR[0]保存返回状态: 0: ARM, 1: thumb             PC指向子程序头地址         可以直接用BL指令也可以用多条指令合成             ARM状态下：MOV LR, PC; BX R4;

#### 3. GDB调试

概要  
    coredump是linux原生的概念，目前有很多工具可以支持coredump调试。其中gdb/trace32都可以支持。  
    coredump包含进程空间的内存，如果在加上含有调试信息的lib/程序，那么可以还原出当时异常的场景，这时你可以查看寄存器内容，调用栈，变量和内存等等。这对分析问题非常有帮助。

    下面我们会一一介绍gdb和trace32如何调试coredump。  
1. gdb (GNU debugger)  
(1). 概述  
    GDB是GNU开源组织发布的一个强大的UNIX下的程序调试工具  
    官网: <http://www.gnu.org/software/gdb/> (可以下载到工具和文档)  
(2). 功能  
    启动或连接程序，可以按照你的自定义的要求随心所欲的运行程序。  
    可让被调试的程序在你所指定的调置的断点处停住。（断点可以是条件表达式）。  
    当程序被停住时，可以检查此时你的程序中所发生的事。  
    动态的改变你程序的执行环境。  
(3). 调试方法  
    在线调试（需要在eng build版本）  
        1. 可以用gdb直接启动一个程序调试。  
        2. 对一个已经运行的程序用gdb attach调试。  
    离线调试（借助coredump）  
        就是本章节的重点。  
(4). 工具来源  
    Android NDK: <http://developer.android.com/sdk/ndk/index.html>。  
    在ndk安装目录下的toolchains/arm-linux-androideabit-$version/prebuilts/linux-x86/bin/arm-linux-androideabi-gdb  
    另外还可以从codebase里找到，在alps/prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-$version/bin/arm-linux-androideabi-gdb  
    【注意】不要使用PC预装的gdb，android有对gdb做修改。  
2. 离线调试  
    gdb离线调试coredump，一定要有这个程序的symbols文件才行。你可以这样启动gdb：  
        arm-linux-androideabi-gdb <program> PROCESS\_COREDUMP  
    gdb有一些启动参数可以配置：  
        -symbols <file>或-s <file>: 从指定文件中读取符号表。  
        -se file: 从指定文件中读取符号表信息，并把他用在可执行文件中。  
        -core <file>或-c <file>: 调试时core dump的core文件。  
        -directory <directory>或-d <directory>: 加入一个源文件的搜索路径。默认搜索路径是环境变量中PATH所定义的路径。  
    当然了，简单使用gdb的话，直接arm-linux-androideabi-gdb即可启动，然后在里面输入对应的命令。  
    进入gdb之后，会有类似shell的窗口，你可以输入任何gdb的命令，gdb的调试就是在这样的窗口上进行的。下一章节我们会介绍gdb常用的几个命令。  
3. 命令  
    gdb的命令可以使用help命令来查看，如下所示  
        命令有很多，gdb把之分成许多个种类。help命令只是例出gdb的命令种类，如果要看种类中的命令，可以使用help <class>命令，如：help breakpoints，查看设置断点的所有命令。也可以直接help <command>来查看命令的帮助。  
    在gdb中输入命令时，可以不用打全命令，只用打命令的前几个字符就可以了，当然命令的前几个字符应该要标志着一个唯一的命令，在Linux下，你可以敲击两次TAB键来补齐命令的全称，如果有重复的，那么gdb会把其例出来。  
    以下列出常用命令（更多的命令请查看gdb官网的用户手册）：

|  |  |  |
| --- | --- | --- |
| 命令 | 缩写 | 描述 |
| backtrace | bt | Prints a stack trace |
| display |  | Displays the value of an expression every time execution stops |
| finish |  | Runs to the end of the function and displays return values of that function |
| jump |  | Jumps to an address and continues the execution there |
| list | l | Lists the next 10 lines |
| next | n | Steps to the next machine language instruction |
| print | p | Prints the value of an expression |
| run | r | Runs the current program from the start |
| set |  | Changes the value of a variable |

(1). 查看栈信息  
    当程序被停住了，你需要做的第一件事就是查看程序是在哪里停住的。  
    backtrace或bt <n>  
        无n: 打印当前的函数调用栈的所有信息。  
        n是一个正整数: 只打印栈顶上n层的栈信息。  
        n是一个负整数: 只打印栈底下|n|层的栈信息。  
    frame <n>或f <n> (n是一个从0开始的整数，是栈中的层编号)  
        如果你要查看某一层的信息，你需要在切换当前的栈，一般来说，程序停止时，最顶层的栈就是当前栈。  
    up <n>或down <n>  
        向栈的上面/下面移动n层，可以不打n，表示向上/下移动一层。  
    info frame或info f  
        打印出更为详细的当前栈层的信息（目前的函数是由什么样的程序语言写成的、函数参数地址及值、局部变量的地址等等）。  
    info args  
        打印出当前函数的参数名及其值。  
    info locals  
        打印出当前函数中所有局部变量及其值。  
(2). 查看源程序  
    显示源代码list或l，list后面可以跟以下参数：  
        <linenum> 行号。  
        <+offset> 当前行号的正偏移量。  
        <-offset> 当前行号的负偏移量。  
        <filename:linenum> 哪个文件的哪一行。  
        <function> 函数名。  
        <filename:function> 哪个文件中的哪个函数。  
        <\*address> 程序运行时的语句在内存中的地址。  
    指定源文件的路径:

        directory <dirname ... >或dir <dirname ... >  
            加一个源文件路径到当前路径的前面，如有多个路径，UNIX下你可以使用":"，Windows下你可以使用";"。  
        directory  
            清除所有的自定义的源文件搜索路径信息。  
        show directories  
            显示定义了的源文件搜索路径。  
(3). 查看变量  
    在你调试程序时，当程序被停住时，你可以使用print(p)命令查看变量  
    print [/<f>] <expr>  
        <expr>是表达式，是你所调试的程序的语言的表达式（GDB可以调试多种编程语言）。  
            可以是当前程序运行中的const常量、变量、函数等（不能用宏）。  
            有几种操作符，它们可以用在任何一种语言中。  
                :: 指定一个在文件或是一个函数中的变量，比如'file'::variable，function::variable。  
                {<type>} <addr> 表示一个指向内存地址<addr>的类型为type的一个对象。  
            如果你的程序编译时开启了优化选项，那么在用GDB调试被优化过的程序时，可能会发生某些变量不能访问，或是取值错误码的情况。  
                gcc可以加-gstabs解决此问题。  
        <f>是输出的格式  
            x 按十六进制格式显示变量。  
            d 按十进制格式显示变量。  
            u 按十六进制格式显示无符号整型。  
            o 按八进制格式显示变量。  
            t 按二进制格式显示变量。  
            c 按字符格式显示变量。  
            f 按浮点数格式显示变量。  
(4). 查看内存  
    examine(x)命令查看内存: x/<n/f/u> <addr>  
        n 是一个正整数，表示显示内存的长度，也就是说从当前地址向后显示几个地址的内容。  
        f 表示显示的格式。如果地址所指的是字符串，那么格式可以是s，如果指令地址，那么格式可以是I。  
        u 表示从当前地址往后请求的字节数，如果不指定的话，默认是4个bytes。也可以用下面的字符来代替，b表示单字节，h表示双字节，w表示四字节，g表示八字节。  
(5). 查看寄存器和线程  
    info registers [<regname ...>]或info reg [<regname …>]  
        查看寄存器的情况（除了浮点寄存器）。  
        <regname ...>查看所指定的寄存器的情况。  
        同样可以使用print命令来访问寄存器的情况，只需要在寄存器名字前加一个$符号就可以：p $sp。  
    info all-registers  
        查看所有寄存器的情况（包括浮点寄存器）。  
    info thread  
        查看所有thread。  
    thread <threadnum>  
        切换到某个thread。  
4. 例子  
    以某个app为例子，发生了NE，产生了db。先用GAT解开db，拿到PROCESS\_COREDUMP，放到/home/db/  
    假设：  
        gdb路径：/home/alps/prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-4.8/bin/arm-linux-androideabi-gdb。  
        symbols路径：/home/alps/out/target/product/$proj/symbols。  
    开始启动gdb，app对应的程序为system/bin/app\_process：  
        cd /home/db/  
        /home/alps/prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-4.8/bin/arm-linux-androideabi-gdb /home/alps/out/target/product/$proj/symbols/system/bin/app\_process PROCESS\_COREDUMP  
    这样就进入gdb的命令行了，下面需要设置symbols搜索路径，设置后gdb会自动加载所需的lib库：  
        set solib-absolute-prefix /home/alps/out/target/product/$proj/symbols  
        set solib-search-path /home/alps/out/target/product/$proj/symbols/system/lib  
    到这里就完成了gdb的启动和加载，之后就可以自由使用各种命令分析NE了，比如bt，info registers等。更多使用gdb的命令或技巧请多查看官方文档或网络上有关gdb的技术分享。

#### trace32调试

Trace32有专门的文章介绍，请参考：

* [MediaTek On-Line](https://online.mediatek.com/)> [Quick Start](https://online.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> Trace32使用教程
* 《Trace32使用教程》里有NE分析章节，里面有讲解NE cmm脚本。

扩展篇: 编译与加载

#### native编译

#### **概要**     从前面的章节我们了解到，Android native程序都是用arm-linux-androideabi-gcc编译的。     了解gcc如何将\*.c/cpp编译成\*.o再将其链接为可执行程序或/lib库，有助于我们将native从编译/加载/执行到崩溃一条路贯通起来。     Android的Makefile只需要将source file填入LOCAL\_SRC\_FILES，然后include $(BUILD\_SHARED\_LIBRARY)或$(BUILD\_EXECUTABLE)就可以将\*.c/cpp/s编译为动态库或可执行程序。其中编译系统做了很多工作，我们不会介绍其中的原因，想要了解的话应该看build相关的文档。     本章节会讲解gcc部分参数的原理及链接的过程。 **1. 编译为obj**     在build/core/definitions.mk有定义transform-c-or-s-to-o-no-deps和transform-cpp-to-o，分别将每个\*.c/s和\*.cpp编译成\*.o，里面传了很多参数给gcc，其中-fpic -fPIE和-fstack-protector是下面会讲解的。 (1). -fpic -fPIE     PIC是Position-Independent Code的缩写，经常被用在共享库中，这样就能将相同的库代码为每个程序映射到一个位置，不用担心覆盖掉其他程序或共享库。     PIE是Position-Independent-Executable的缩写，只能应用在可执行程序中。PIE和PIC很像，但做了一些调整（不用PLT，使用PC相关的重定位）。-fPIE给编译用，-pie给链接(ld)用。     例如，一个程序没有使用PIC被链接到0地址，那么系统将其加载到0地址，程序可以正常运行：

#### 

#### 如果系统将其加载到0x8000，程序会异常：

#### 

#### 如果开启PIC，这会是这样：

#### 

使用了什么技术可以达到这样呢？

    a. 对于访问自己的函数等可以使用以PC+偏移量的方式达到效果：

#### 

#### **b.**而访问外部共享库等需要GOT/PLT支持才行。         PLT(Procedure Linkage Table): 由一个个桩函数组成的跳转表         GOT(Global Offset Table): 由一个个偏移量组成的偏移表，决定PLT跳转的位置（由linker修改）     要调用外部的函数会先跳转到PLT，PLT再从GOT获取实际的位置，再跳转过去。PLT/GOT类似一座桥：

#### 

#### **实例分析**：     某个程序的函数xxx()调用外部函数\_\_futex\_wait()，xxx()在.text段里会直接跳转到\_\_futex\_wait的PLT里，PLT会读取对应GOT的地址，直接跳转过去：

#### 

#### GOT表的地址会在linker加载库时被修改，因为所有的库都是由linker加载的，也只有linker知道\_\_futex\_wait()被加载到哪里去了。     这里linker会有两种做法：        **第1**：在加载库时就把GOT全部修改好，后面程序调用就无需修改，加快了程序的运行，但启动较慢。这个就是所谓立即绑定，Android目前使用这种方法。         **第2**：先不改GOT，此时GOT默认会指向第1个PLT，这个PLT会比较特殊，linker会修改这个PLT对应的GOT，使它指向linker某个函数，那这样只有GOT没有被修改下会自动跳转到linker，由linker再去单独修改这个GOT，达到按需修改，相比第1种，可以快速启动，但可能影响运行。这种也称为迟绑定技术（Lazy binding）。     大家看到这里肯定会想，方法有了，那linker将GOT修改为什么值呢？其实在动态库里有.rel.dyn和.rel.plt段，里面包含了哪里要修改，修改的方法和对应的符号，linker就可以轻松完成这件艰巨的任务：

#### 

#### (2). -fstack-protector     顾名思义就是保护堆栈，每一个函数在运行时都有自己的栈帧，如果代码没有写好，很可能将自己甚至是其他的栈帧踩坏，那如何防护呢？简单的方法就是在栈帧头部也就是在局部变量开始之前多存储一个\_\_stack\_chk\_guard值，用于在函数返回前取出来和\_stack\_chk\_guard做对比，失败则调用\_\_stack\_chk\_fail函数，这个就是该参数完成的行为。     以下是示意图：

#### 

#### 

打开该功能后，编译会自动插入所需代码：

#### 

#### 当然该参数不是所有的函数都保护的，一般踩坏栈帧多半是for/while循环迭代数组，所以可以如下配置：         -fstack-protector：只保护有定义局部数组且数组个数大于8的函数。         --param=ssp-buffer-size=xx：在-fstack-protector基础上增加该参数可以修改数组个数。         -fstack-protector-all：保护所有函数。     发生stack corruption时会跑到\_\_stack\_chk\_fail()，印出相关信息：

#### 

2. 静态链接  
    build/core/combo/TARGET\_linux-arm.mk里有定义transform-o-to-static-executable-inner，将\*.o链接成静态可执行程序，静态可执行程序是一个完整的程序，不需要额外的共享库即可执行，比如/init,/sbin/adbd等。  
    链接器用的是arm-linux-androideabi-g++，主要的参数和介绍如下（这里以test.c编译为test为例子）：  
        -Bstatic ##表示静态链接##  
        out/target/product/$proj/obj/lib/crtbegin\_static.o  
        out/target/product/$proj/obj/EXECUTABLES/test\_intermediates/test.o  
        prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-$version/lib/gcc/arm-linux-androideabi/$version-google/armv7-a/libgcc.a  
        out/target/product/$proj/obj/lib/crtend\_android.o  
    大家注意到没有，除了test.o居然还链接了crtbegin\_static.o/libgcc.a和crtend\_android.o，这是怎么回事？？

    libgcc.a后面会详细讲，而crtbegin\_static.o/crtend\_android.o对应代码是（从bionic/libc/Andoird.mk看出）bionic/libc/arch-arm/bionic目录下的crtbegin.c和crtend.s，其中定义了.preinit\_array，.init\_array和.fini\_array段以及\_start函数。  
    再查看gcc默认的链接脚本（prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-$version/arm-linux-androideabi/lib/ldscripts/armelf\_linux\_eabi.x）：

#### 

#### 里面定义了入口函数是\_start()（注意不是main()!!!），以及加载位置是0x8000。其中crtbegin.c至关重要，main()函数在\_\_libc\_init()里调用到，而之前已经做了很多事，这也就是crt（c-runtime，c运行环境）的工作。 **3. 动态链接**     build/core/combo/TARGET\_linux-arm.mk里有定义transform-o-to-executable-inner和transform-o-to-shared-lib-inner，分别将\*.o链接为动态可执行程序和共享库。动态可执行程序需要linker才能进一步运行的。链接器也是用arm-linux-androideabi-g++，下面分析讲解： (1). 动态可执行程序     主要的参数和介绍如下:         -Bdynamic ##表示动态链接##         -fPIE ##链接为位置无关##         -pie         -Wl,-z,now ##表示立即绑定##         -Wl,-dynamic-linker,/system/bin/linker ##指定解释器##         -lc -lstdc++ -lm ##所需的lib库，需要显式指定，否则编译报错##         out/target/product/$proj/obj/lib/crtbegin\_dynamic.o         out/target/product/$proj/EXECUTABLES/test\_intermediates/test.o         prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-$version/lib/gcc/arm-linux-androideabi/$version-google/armv7-a/libgcc.a         out/target/product/$proj/obj/lib/crtend\_android.o     相比静态链接，多了PIE，立即绑定，解释器。crtbegin\_dynamic.o和crtbegin\_static.o一样。另外的差别是加载地址可以有系统决定，不像静态链接程序固定到0x8000。 (2). 共享库     主要的参数和介绍如下:         -Wl,-shared,-Bsymbolic ##编译为共享库##         -Wl,-z,now ##表示立即绑定##         -lc -lstdc++ -lm ##该lib所需的lib库，需要显式指定，否则编译报错##         out/target/product/$proj/obj/lib/crtbegin\_so.o         out/target/product/$proj/EXECUTABLES/test\_intermediates/test.o         prebuilts/gcc/linux-x86/arm/arm-linux-androideabi-$version/lib/gcc/arm-linux-androideabi/$version-google/armv7-a/libgcc.a         out/target/product/$proj/obj/lib/crtend\_so.o     相比动态可执行程序，没有太大变化，不过crtbegin\_so.o和crtend\_so.o少了\_start()等相关代码，共享库不需要入口函数。 **4. 移除调试信息**     链接后的程序一般都比较大，因为包含了调试信息，而这些信息不会在运行时用到，所以会将其删除后在放入手机里。这里用到了arm-linux-androideabi-strip --strip-all功能：

#### 

5. libgcc.a  
    GCC在一些平台上提供了一个低级运行时库，libgcc.a或者libgcc\_s.so.1. 一旦需要执行某些过于复杂而无法通过内嵌代码实现的操作，GCC便会自动生成对这些库函数的调用。  
    大多数libgcc中的函数用来处理目标处理器不能直接执行的算术运算（包括整数乘除，所有浮点运算）, 还包括异常处理, 少数杂项操作。  
    详细请参考: <http://gcc.gnu.org/onlinedocs/gccint/Libgcc.html#Libgcc>  
6. crt\*.o  
    这些都是c运行时库，用于执行进入main之前的初始化和退出main之后的扫尾工作：  
        \_\_PREINIT\_ARRAY\_\_, \_\_INIT\_ARRAY\_\_:初始化时调用的函数数组  
        \_\_FINI\_ARRAY\_\_: 结束时调用的函数数组  
        \_\_CTOR\_LIST\_\_: 初始化时调用的构造函数数组

    这些数组都是以-1开始，以0结束：

    对应的代码（crtbegin.c）如下：

#### 

结语

    上面只是讲解编译涉及到的部分知识，实际上用到的技术更多，需要大家自己深入了解和实践。

#### ELF/coredump结构

#### **历史**     ELF-可执行链接格式最初是由UNIX系统实验室(USL)作为应用程序二进制接口(ABI)开发和发行。     工具接口标准委员会TIS已经将ELF作为运行在Intel32位架构之上的各类型操作系统的可导出对象文件格式标准。     ELF标准为开发者提供了一组横跨多运行环境的二进制接口定义来组织软件开发。 **1. 类型** (1). 定义（目标文件格式主要三种）     可重定向文件(Relocatable file): 文件保存着代码和适当的数据，用来和其他的目标文件一起来创建一个可执行文件或者是一个共享目标文件。由编译器和汇编器生成，将由链接器处理。     可执行文件(Executable File): 文件保存着一个用来执行的程序；该文件指出了exec如何来创建程序进程映象。所有重定向和符号都解析完成了，如果存在共享库的链接，那么将在运行时解析。     共享目标文件(Shared object file): 就是所谓的共享库。文件保存着代码和合适的数据，用来被下面的两个链接器链接。第一个是连接编辑器[ld]，可以和其他的可重定向和共享目标文件来创建其他的目标文件。第二个是动态链接器，联合一个可执行文件和其他的共享目标文件来创建一个进程映象。         包含链接时所需的符号信息和运行时所需的代码。

#### 

2). 识别  
    通过file命令可以看到哪种类型的ELF，比如：  
        gcc -c helloworld.c  
        file helloworld.o  
        helloworld.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped ##可重定向文件##

        gcc -o helloworld helloworld.o  
        file helloworld  
        helloworld: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, not stripped ##可执行文件##  
  
        file libc.so  
        libc.so: ELF 32-bit LSB shared object, ARM, version 1 (SYSV), dynamically linked, not stripped ##共享目标文件##

#### **2. 格式**     目标文件既要参与程序链接又要参与程序执行。出于方便性和效率考虑，目标文件格式提供了两种并行视图，分别反映了这些活动的不同需求。编译器，链接器把它看作是sections的集合，loader把它看作是segments的集合：

#### 

#### 段和节是存在包含关系：

#### 

#### 

#### **3. 结构** (1). 索引表     一般的ELF文件包括三个索引表：        **ELF header**             在文件的开始，保存了路线图(road map)，描述了该文件的组织情况。

#### 

#### 用readelf查看elf头部信息如下：

#### 

#### **Section header table**             包含了描述文件节区的信息，每个节区在表中都有一项，每一项给出诸如节区名称、节区大小这类信息。用于链接的目标文件必须包含节区头部表，其他目标文件可选。             ELF header中的e\_shoff变量就是保存section header table入口对文件头的偏移量。

#### 

  通过Section header table，我们可以定位到所有的节。

        Program header table  
            告诉系统如何创建进程映像。用来构造进程映像的目标文件必须具有程序头部表，可重定位文件不需要这个表。  
            ELF header中的e\_phoff变量就是保存program header table入口对文件头的偏移量。

#### 

通过program header table，我们可以定位到所有的段。

(2). Section  
    在ELF spec里有预定义了几个节结构：

#### 

#### 这边只介绍下以下节：     **a.**symbtab和dynsym节         用于存储符号名字，值，类型等信息：

#### 

#### 类型可以表示函数/变量等：

#### 

#### 使用readelf导出符号表：

#### 

#### **b.** init/fini/ctors节         在讲crt章节有介绍到。.preinit\_array, .init\_array, .fini\_array, .ctors节结构是一样的，都是函数指针数组（开始于-1, 结束于0）

#### 

#### 如何将一个函数添加到这些节当中呢？你可以这样：

#### 

添加到.init\_array的函数将在main()函数前就开始执行了，在bionic有些函数使用了这样的手段，大家可以将它们找出来。  
        全局C++类的构造函数也会被添加到\_\_INIT\_ARRAY\_\_中。编译器会额外生成一个函数，该函数将文件中的所有全局类调用构造函数，并将析构函数注册到atexit函数。

    c. Debug节  
        采用DWARF标准存储调试信息，我们分析NE都需要这些调试信息，这些信息会被strip掉，因此手机里的程序/库的ELF没有这部分信息，只有symbols目录下的才有（所以一定要保留symbols目录！）。  
        debug节有很多种类，比如：  
            .debug\_line：存放代码行号和地址对应信息。  
            .debug\_frame：存放解栈信息。  
            .debug\_loc：存放函数里的寄存器代表符号的信息。  
            .debug\_pubnames：存放全局变量及函数。  
        这些节在调试中怎么体现出来呢？使用gdb/trace32，你将看到这些节带来的信息：

#### 

(3). segment  
    段也有不同的类型，有些段中保存着机器指令，有些保存着已初始化的变量，有些则作为进程镜像的一部分被操作系统读入内存。

    类型如下：

#### 

#### 一般一个完整的共享库或可执行程序存在2个PT\_LOAD（该类型会被加载到内存），用于保存代码段和数据段。

#### 

4. coredump结构  
    coredump只是其中一种ELF，在经过前面的学习，应该很容易理解coredump结构。

    coredump是ELF第4种结构，仅为调试而存在：

#### 

#### coredump没有节，只有段，而且只有两种段类型，PT\_NOTE和PT\_LOAD，note段保存了现场寄存器，线程和引起崩溃信号的信息，而PT\_LOAD这是进程空间内存数据段。

#### 

#### PT\_NOTE结构如下：

#### 

#### 其中的type可以是：

#### 

  其中NT\_PRSTATUS就记录了CPU寄存器信息。

结语

    以上只是抛砖引玉，真正全面的资料大家还需参考ELF标准和ARM对ELF的扩展。

#### ELF加载执行

#### **1. ELF loader**     这个ELF loader指的是kernel elf loader。可执行程序以ELF文件的形式保存在文件系统中，核心的load\_elf\_binary会首先将其映像文件映射到内存，然后映射并执行其解释器(linker)的代码。linker的代码段是进程间共享的，但数据段为各进程私有，linker执行完后会自动跳转到目标映像的入口地址：         sys\_execve() => do\_execve() => search\_binary\_handler() => load\_elf\_binary() (1). 加载     在load\_elf\_binary()函数里，会解析这个程序的ELF结构，判断是否为ELF，然后根据PT\_LOAD段加载到对应的进程空间内：

#### 

#### 如果需要linker，linker也会被系统加载。

#### 

#### (2). 初始化栈     分配好栈后，将参数压入栈中，然后是环境变量，最后是辅助向量。

#### 

这里介绍下辅助向量，辅助向量会传递内核一些信息，比如CPU型号和功能，页面大小等。

    结构如下：

#### 

(3). 准备调度

    做完之后，就可以等待调度了。

2. linker  
    Kernel加载完可执行程序和对应的连接器后，就跳转到用户空间的入口地址(ICS/GB:0xB0001000, JB之后:系统动态分配)开始执行。  
    Android的加载/链接器linker主要用于实现共享库的加载与链接（支持应用程序对库函数的隐式和显式调用）：  
        隐式调用  
            应用程序的编译与静态库大致相同，只是在静态链接的时候通过--dynamic-linker /system/bin/linker指定动态链接器（该信息将被存放在ELF文件的.interp节中，内核执行目标映像文件前将通过该信息加载并运行相应的解释器程序linker）并链接相应的共享库。  
            与ld.so不同的是，Linker目前没有提供Lazy Binding机制，所有外部过程引用都在映像执行之前解析。

        显式调用  
            可以通过linker中提供的接口dlopen()，dlsym()，dlerror()和dlclose()来动态加载和链接共享库。  
   linker里面涉及的加载和其他平台大同小异，大家可以查看相关书籍。

#### 编程篇: linux c编程

#### 信号

#### **1. 定义**     简而言之，信号是一种软件中断，提供了一种处理异步的方法，信号发生是随机的。例如键盘输入中断按键(^C)，它的发生在程序执行过程中是不可预测的。     硬件异常也能产生信号，例如被零除、无效内存引用（test里产生的就是这种错误）等。这些条件通常先由内核硬件检测到，然后通知内核。内核将决定产生什么样的信号。     同一个信号的额外发生通常不会被排队。如果信号在被阻塞时发生了5次，当我们反阻塞这个信号时，这个信号的信号处理函数通常只被调用一次。     同一时刻只能处理一个信号，在信号处理函数发信号给自己时，该信号会被pending。     信号的数值越小，则优先级越高。当进程收到多个待处理信号时，总是先处理优先级别高的信号。     信号处理函数的栈可以使用被中断的也可以使用独立的，具体可以通过系统调用设置。 **2. 处理方式**     忽略：接收到信号后不做任何反应。     捕获：用自定义的信号处理函数来执行特定的动作。     默认：接收到信号后按系统默认的行为处理该信号。             这是多数应用采取的处理方式

3. 类型

    这里列出主要的信号，具体可查看bionic/libc/kernel/arch-arm/asm/signal.h:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 名称 | 数字 | 标准 | 默认行为 | 说明 |
| SIGILL | 4 | ANSI | 终止+coredump | 执行了非法指令. 通常是因为可执行文件本身出现错误, 或者试图执行数据段. 堆栈溢出时也有可能产生这个信号 |
| SIGABRT | 6 | ANSI | 终止+coredump | 调用abort函数生成的信号 |
| SIGBUS | 7 | 4.2 BSD | 终止+coredump | 非法地址, 包括内存地址对齐(alignment)出错。比如访问一个四个字长的整数, 但其地址不是4的倍数。它与SIGSEGV的区别在于后者是由于对合法存储地址的非法访问触发的(如访问不属于自己存储空间或只读存储空间) |
| SIGFPE | 8 | ANSI | 终止+coredump | 在发生致命的算术运算错误时发出. 不仅包括浮点运算错误, 还包括溢出及除数为0等其它所有的算术的错误 |
| SIGSEGV | 11 | ANSI | 终止+coredump | 试图访问未分配给自己的内存, 或试图往没有写权限的内存地址写数据。访问空指针，野指针基本都产生这个信号，也是最常见的信号 |
| SIGSTKFLT | 16 | N/A | 终止 | 堆栈错误 |
| SIGPIPE | 13 | POSIX | 终止 | 管道破裂。这个信号通常在进程间通信产生，比如采用FIFO(管道)通信的两个进程，读管道没打开或者意外终止就往管道写，写进程会收到SIGPIPE信号。此外用Socket通信的两个进程，写进程在写Socket的时候，读进程已经终止 |
| SIGTRAP | 5 | POSIX | 终止+coredump | 由断点指令或其它trap指令产生. 由debugger使用 |
| SIGHUP | 1 | POSIX | 终止 | 用户终端连接(正常或非正常)结束时发出, 通常是在终端的控制进程结束时, 通知同一session内的各个作业, 这时它们与控制终端不再关联 |
| SIGINT | 2 | ANSI | 终止 | 程序终止(interrupt)信号, 在用户键入INTR字符(通常是Ctrl-C)时发出，用于通知前台进程组终止进程 |
| SIGQUIT | 3 | POSIX | 终止+coredump | 和SIGINT类似, 但由QUIT字符(通常是Ctrl-\)来控制. 进程在因收到SIGQUIT退出时会产生core文件, 在这个意义上类似于一个程序错误信号 |
| SIGKILL | 9 | POSIX | 终止 | 用来立即结束程序的运行. 本信号不能被阻塞、捕获和忽略。如果管理员发现某个进程终止不了，可尝试发送这个信号 |
| SIGCHLD | 17 | POSIX | 忽略 | 子进程结束时, 父进程会收到这个信号。如果父进程没有处理这个信号，也没有等待(wait)子进程，子进程虽然终止，但是还会在内核进程表中占有表项，这时的子进程称为僵尸进程。这种情 况我们应该避免(父进程或者忽略SIGCHILD信号，或者捕捉它，或者wait它派生的子进程，或者父进程先终止，这时子进程的终止自动由init进程来接管) |
| SIGCONT | 18 | POSIX | 继续/忽略 | 让一个停止(stopped)的进程继续执行. 本信号不能被阻塞. 可以用一个handler来让程序在由stopped状态变为继续执行时完成特定的工作. 例如, 重新显示提示符.. 在进程挂起时是继续，否则是忽略 |
| SIGSTOP | 19 | POSIX | 暂停 | 暂停进程的执行. 注意它和terminate以及interrupt的区别:该进程还未结束, 只是暂停执行. 本信号不能被阻塞、捕获或忽略 |
| SIGALRM | 14 | POSIX | 终止 | 时钟定时信号, 计算的是实际的时间或时钟时间. alarm函数使用该信号 |

#### 【TIPS】: 编号为0的信号，用以测试进程是否拥有信号发送的权限，并不会被实际发送

4. 详细信息

    kernel可以传递更多信号相关的信息给native层，通过ptrace或waitid函数可以获取信号详细信息。

    信息保存在siginfo结构体，如下：

#### 

下表展示了各种信号的si\_code值，由SUS(Single UNIX Specification:是POSIX.1标准的超集)定义：

    SIGSEGV的代码和原因：

|  |  |
| --- | --- |
| 代码 | 原因 |
| SEGV\_MAPERR: 1 | 地址没有映射到对象（大部分的异常是这种类型） |
| SEGV\_ACCERR: 2 | 映射的对象的无效权限 |

    SIGBUS的代码和原因：

|  |  |
| --- | --- |
| 代码 | 原因 |
| BUS\_ADRALN: 1 | 地址不对齐(自然对齐) |
| BUS\_ADRERR: 2 | 不存在的物理地址 |
| BUS\_OBJERR: 3 | 对象指定的硬件错误 |

    SIGFPE的代码和原因：

|  |  |
| --- | --- |
| 代码 | 原因 |
| FPE\_INTDIV: 1 | 整数被0除 |
| FPE\_INTOVF: 2 | 整数溢出 |
| FPE\_FLTDIV: 3 | 浮点数被0除 |
| FPE\_FLTOVF: 4 | 浮点数溢出 |
| FPE\_FLTUND: 5 | 浮点数下溢 |
| FPE\_FLTRES: 6 | 浮点数不精确结果 |
| FPE\_FLTINV: 7 | 无效的浮点数操作 |
| FPE\_FLTSUB: 8 | 范围外的下标 |

    SIGILL的代码和原因：

|  |  |
| --- | --- |
| 代码 | 原因 |
| ILL\_ILLOPC: 1 | 违法操作码 |
| ILL\_ILLOPN: 2 | 违法操作数 |
| ILL\_ILLADR: 3 | 违法地址模式 |
| ILL\_ILLTRP: 4 | 违法陷阱 |
| ILL\_PRVOPC: 5 | 特权操作码 |
| ILL\_PRVREG: 6 | 特权寄存器 |
| ILL\_COPROC: 7 | 协进程错误 |
| ILL\_BADSTK: 8 | 内部栈错误 |

    SIGTRAP的代码和原因：

|  |  |
| --- | --- |
| 代码 | 原因 |
| TRAP\_BRKPT: 1 | 进程中断点陷阱 |
| TRAP\_TRACE: 2 | 进程跟踪陷阱 |

    SIGCHLD的代码和原因：

|  |  |
| --- | --- |
| 代码 | 原因 |
| CLD\_EXITED: 3 | 子进程已经退出 |
| CLD\_KILLED: 4 | 子进程已异常退出(无coredump) |
| CLD\_DUMPED: 5 | 子进程已异常退出(有coredump) |
| CLD\_RAPPED: 6 | 跟踪的子进程已经被套住 |
| CLD\_STOPPED: 7 | 子进程被停止 |
| CLD\_CONTINUED: 8 | 停止的子进程被继续 |

    SIGPOLL的代码和原因：

|  |  |
| --- | --- |
| 代码 | 原因 |
| POLL\_IN: 1 | 数据可以被读 |
| POLL\_OUT: 2 | 数据可以被写 |
| POLL\_MSG: 3 | 输入消息可用 |
| POLL\_ERR: 4 | I／O错误 |
| POLL\_PRI: 5 | 高优先级消息可用 |
| POLL\_HUP: 6 | 设备断开连接 |

#### socket/ptrace

1. 1. socket  
   (1). 介绍  
       使用套接字除了可以实现网络间不同主机间的通信外，还可以实现同一主机的不同进程间的通信，且建立的通信是双向的通信。  
   (2). 服务器端建立 (socket -> bind -> listen -> accept -> read/write)  
       程序通过调用socket函数(AF\_LOCAL, SOCK\_STREAM)，建立了主动连接的套接字.  
       调用bind函数，将套接字与地址信息关联起来.  
       调用listen函数实现对该端口的监听，同时变为监听套接字.  
       当有连接请求时，通过调用accept函数建立与客户端的连接.  
       调用read/write函数来读取/发送消息，当然也可以使用recv/send函数实现相同的功能.  
   (3). 客户端建立 (socket -> connect -> read/write)  
       程序通过调用socket函数(AF\_LOCAL, SOCK\_STREAM) ，建立了主动连接的套接字。  
       调用connect函数(附带地址信息)，向服务器端发出连接请求。  
       调用read/write函数来读取/发送消息，当然也可以使用recv/send函数实现相同的功能。  
   2. ptrace (process trace)  
   (1). 介绍  
       ptrace()是个系统调用，提供了一种父进程可以控制子进程运行，并可以检查和改变它的核心image (用于实现断点调试，代码分析)  
           修改被跟踪进程的空间(内存或寄存器)。  
           任何传递给被跟踪进程的信号(除了SIGKILL)都会使得这个进程进入暂停状态，这时跟踪进程通过wait()得知相关的状态并做相应的修改。
2. (2). 限制  
       不能跟踪进程init。不能跟踪自己。  
   (3). 功能

#### 

#### PTRACE\_TRACEME         本进程被其父进程所跟踪。其父进程应该希望跟踪子进程，用于debugger。     PTRACE\_ATTACH         跟踪指定pid 进程,成为pid的父进程,并停止pid进程。     PTRACE\_DETACH         结束跟踪。     PTRACE\_PEEKTEXT, PTRACE\_PEEKDATA和PTRACE\_POKETEXT, PTRACE\_POKEDATA         读取/修改被跟踪进程的user space里的内存。     PTRACE\_GETREGS, PTRACE\_SETREGS、PTRACES\_GETFPREGS, PTRACE\_SETFPREGS、PTRACE\_GETVFPREGS, PTRACE\_SETVFPREGS         读取/修改被跟踪进程的通用/浮点寄存器值。     PTRACE\_SYSCALL, PTRACE\_CONT         重新运行，(PTRACE\_SYSCALL会使每次系统调用暂停)。     PTRACE\_SINGLESTEP         设置单步执行标志。     PTRACE\_KILL         杀掉子进程，使它退出。

#### libc功能模块

#### **1. libc** (1). libc是Standard C library的简称，它是符合ANSI C标准的一个函数库。     libc库提供C语言中所使用的宏，类型定义，字符串操作函数，数学计算函数以及输入输出函数等。     正如ANSI C是C语言的标准一样，libc只是一种函数库标准，每个操作系统都会按照该标准对标准库进行具体实现。     通常我们所说的libc是特指某个操作系统的标准库，比如我们在Linux操作系统下所说的libc即glibc。glibc是类Unix操作系统中使用最广泛的libc库，它的全称是GNU C Library。 (2). 类Unix操作系统通常将libc库作为操作系统的一部分 (被视为操作系统与用户程序之间的接口)

#### 

#### libc库不仅实现标准C语言中的函数，而且也包含自己所属的函数接口。比如在glibc库中，既包含标准C中的fopen()，又包含类Unix系统中的open()。在类Unix操作系统中，如果缺失了标准库，那么整个操作系统将不能正常运转。在Android也是一样的：

#### 

#### 而Windows系统并不将libc库作为整个核心操作系统的一部分。通常每个编译器都附属自己的libc库，这些libc既可以静态编译到程序中，又可以动态编译到程序中。也就是说应用程序依赖编译器而不是操作系统。 (3). 封装函数     在Linux系统中，glibc库中包含许多API，大多数API都对应一个系统调用，比如应用程序中使用的接口open()就对应同名的系统调用open()。         glibc库中通过封装例程(Wrapper Routine)将API和系统调用关联起来。         API是头文件中所定义的函数接口，而位于glibc中的封装例程则是对该API对应功能的具体实现。     事实上接口open()所要完成的功能是通过系统调用open()完成的，因此封装例程要做的工作是先将接口open()中的参数复制到相应寄存器中，然后引发一个异常，从而系统进入内核去执行sys\_open()，最后当系统调用执行完毕后，封装例程还要将错误码返回到应用程序中。

#### 

注意】函数库中的API和系统调用并没有一一对应的关系。应用程序借助系统调用可以获得内核所提供的服务，像字符串操作这样的函数并不需要借助内核来实现，因此也就不必与某个系统调用关联。  
    也不是必须通过封装例程才能使用系统调用，syscall()和\_syscallx()两个函数可以直接调用系统调用。

2. pthread

    bionic libc包含了pthread，pthread另有标准定义，大家也可以参考。

3. dlmalloc

    KK及以前版本用的是dlmalloc作为malloc/free的分配器，学习dlmalloc对于堆内存管理是非常必要的。具体资料可以到网络搜索。

结语

    libc是linux/Android最基础的库，掌握它将更好理解NE。

#### 专题篇: 专题分析

#### 1.踩内存专题

踩内存问题一直都困扰着每个Android工程师，调试难度很大，处于项目的各个阶段，严重影响软件品质。我们专门列一个专题分析，详情请看：

* MediaTek On-Line> Quick Start> 踩内存专题分析

#### 2.fd专题

fd泄漏和意外关闭fd引起的逻辑问题在android也是比较常见的问题，经过几代的发展，调试已相当方便，已经形成固定的套路分析了，详情请看：

* MediaTek On-Line> Quick Start> 文件描述符(fd)专题分析

通过这篇专题文章，相信这类问题大家就可以自行处理掉了。

#### 3.售后收集重启专题

售后软件品质已越来越重要，以前通过返修/客退机来分析解决量产前未发现的问题，现在有更好的方法，通过后台自动收集故障信息，然后回传处理分析。我们专门列一个专题分析，详情请看：

* MediaTek On-Line> Quick Start> 售后收集重启专题分析

#### 实例篇: 案例分析

#### 1.移除SD卡后引发SIGBUS NE

问题背景：

移除 SD 卡，打開 contacts 時 100% 發生 com.android.contacts 的 Native Exception

分析过程：

使用GAT解析DB文件，从\_exp\_main.txt看到exception type是SIGBUS，对于SIGBUG类型的NE最常见的一种情况是后备存储器异常（不见了等）导致，比如文件映射进来，结果访问时，文件被删除了。

\_exp\_main.txt：

 Exception Class: Native (NE)  
 Exception Type: SIGBUS  
  
 Current Executing Process:   
 pid: 8016, tid: 8033  
 com.android.contacts

 从\_exp\_detail.txt可查看到具体的fault address：

 pid: 8016, tid: 8033, name: AccountChangeLi >>> com.android.contacts <<<  
 signal 7 (SIGBUS), code 2 (BUS\_ADRERR), fault addr 0x7f910c4000

 从PROCESS\_MAPS文件可知fault address落在：

 7f910c3000-7f910c5000 r--s 02929000 fd:01 12 /mnt/asec/com.facebook.katana-1/base.apk

此目录/mnt 是SD卡的路径。

根本原因：

移除SD卡的行为正是此题NE发生的原因，com.android.contacts需要访问安装在SD卡上的应用，而此时SD卡已经被拔了。

解决方法：

 应用不建议安装在SD卡上，如果是安装在SD卡，请不要随意插拔SD卡。

#### 2.孤儿进程组问题导致的重启

问题背景：

1、安装豌豆荚，QQ后  
2、用豌豆荚下载搜狗输入法，天天德州，天天爱消除，泡泡龙亚特，并运行这些应用  
3、点击豌豆荚时，手机重启

分析过程：

1.查看kernel log看到：  
<7>[ 1884.162500] -(1)[11342:libxguardian.so][name:mtprof&][signal][11342:libxguardian.so] send death sig 1 to [1094:ndroid.systemui:W]  
<7>[ 1884.162984] -(1)[11342:libxguardian.so][name:mtprof&][signal][11342:libxguardian.so] send death sig 1 to [988:system\_server:W]  
<7>[ 1884.163024] -(1)[11342:libxguardian.so][name:mtprof&][signal][11342:libxguardian.so] send death sig 1 to [258:main:W]  
<13>[ 1884.166606] (2)[1:init]init: Untracked pid 11342 exited with status 0  
<7>[ 1884.167235] -(2)[11212:libuuid.so][name:mtprof&][signal][11212:libuuid.so] send death sig 6 to [11212:libuuid.so:R]  
<12>[ 1884.290928] (1)[11335:logd.reader.per]logd: logd.reader.per thread stop.  
<13>[ 1884.313276] (3)[1:init]init: Untracked pid 11280 killed by signal 1  
<13>[ 1884.315574] (1)[1:init]init: Service 'zygote' (pid 258) killed by signal 1  
<13>[ 1884.315630] (1)[1:init]init: Service 'zygote' (pid 258) killing any children in process group  
system\_server和zygote被杀导致android reboot

从以上这段log可以看出是孤儿进程组的问题：  
若父进程退出导致进程组成为孤儿进程组，且该进程组中有进程处于停止状态（收到SIGSTOP或SIGTSTP信号），SIGHUP信号会被发送到该进程组中的每一个进程。  
系统对SIGHUP信号的默认处理是终止收到该信号的进程。所以若程序中没有捕捉该信号，当收到该信号时，进程就会退出。  
孤儿进程组条件:  
– App fork child process.  
– App exit.  
– Child process exit and there is a(or more) process is stop.

关于孤儿进程组的问题也可以参考FAQ：

[FAQ11577]Linux 孤儿进程组问题说明

此笔case中，libxguardian.so是孤儿进程，而同在这个孤儿进程组中的libuuid.so因为NE而处于stop状态，所以SIGHUP信号会被发送到该进程组中的每一个进程，包括zygote，所以android重启了。

2.用GAT解析libuuid.so(豌豆荚) NE的DB

Exception Class: Native (NE)

Exception Type: SIGABRT

backtrace:

 #00 pc 00045464  /system/bin/linker (\_\_dl\_tgkill+12)  
  
#01 pc 00044a1b  /system/bin/linker (\_\_dl\_pthread\_kill+34)  
  
#02 pc 0003cdbf  /system/bin/linker (\_\_dl\_raise+10)  
  
#03 pc 0003b86b  /system/bin/linker (\_\_dl\_\_\_libc\_android\_abort+34)  
  
#04 pc 0003a6c0  /system/bin/linker (\_\_dl\_abort+4)  
  
#05 pc 0003c8bb  /system/bin/linker (\_\_dl\_\_\_libc\_fatal+22)  
  
#06 pc 000093bb  /system/bin/linker (\_\_dl\_\_ZL29\_\_linker\_init\_post\_relocationR19KernelArgumentBlockj+2250)  
  
#07 pc 00008a5b  /system/bin/linker (\_\_dl\_\_\_linker\_init+358)  
  
#08 pc 00002770  /system/bin/linker (\_start+4)

 通过addr2line定位到在以下函数中调用的abort：

linker.cpp中的

 4258  if (elf\_hdr->e\_type != ET\_DYN) {  
  
4259    \_\_libc\_fatal("\"%s\": error: only position independent executables (PIE) are supported.",  
  
4260                 args.argv[0]);  
  
4261  }

 在NE之前的main log中也看到：

F libc    : "/data/user/0/com.wandoujia.phoenix2/lib/libuuid.so": error: only position independent executables (PIE) are supported.

异常的应用不是基于PIE编译的，所以被linker拦截下来了。

PIE安全检查机制是Google的安全机制，从L版本之后，如果调用的可执行文件不是基于PIE方式编译的，则无法运行。

编译时在Android.mk中加入如下flag即是基于PIE方式编译：  
LOCAL\_CFLAGS += -pie -fPIE  
LOCAL\_LDFLAGS += -pie -fPIE

根本原因：

libxguardian.so应用父进程创建子进程,但父进程先退出，这样进程所在的进程组会成为孤儿进程组。

libuuid.so(豌豆荚)应用不是基于PIE编译导致NE

解决方法：

1.更新豌豆荚版本，最新版本并没有这个问题

2.libxguardian.so应用需修改架构

#### 3.符号缺失引起链接失败

问题背景：

打入ViLTE patch后ims注册失败

毕现

分析过程：

从main log看，开机之后VT Thread一直都没有起来，

因为打开ViLTE后，IMS一直在等待VTService来连接，但是一直没有等到，原因就是VTService进程一直起不来

Line 1649: [ 28.391873] (7)[1:init]init: Starting service 'vtservice'...  
 Line 1694: [ 28.512512] (7)[1:init]init: Service 'vtservice' (pid 3079) exited with status 1  
 Line 1695: [ 28.512548] (7)[1:init]init: Service 'vtservice' (pid 3079) killing any children in process group  
 Line 2852: [ 33.076339] (2)[1:init]init: Starting service 'vtservice'...  
 Line 2876: [ 33.338117] (5)[1:init]init: Service 'vtservice' (pid 3643) exited with status 1  
 Line 2877: [ 33.338167] (5)[1:init]init: Service 'vtservice' (pid 3643) killing any children in process group  
 Line 4000: [ 38.091726] (3)[1:init]init: Starting service 'vtservice'...  
 Line 4026: [ 38.196002] (0)[1:init]init: Service 'vtservice' (pid 4142) exited with status 1  
 Line 4027: [ 38.196031] (0)[1:init]init: Service 'vtservice' (pid 4142) killing any children in process group  
 Line 5282: [ 43.201721] (0)[1:init]init: Starting service 'vtservice'...

检查了vtservice都没有问题，那么为何vtservice为自己退出呢？怀疑是vtservice自己的行为，因此加log到main函数，结果一句log都没印出来。

起初以为ALOGI打印log函数被logd屏蔽掉了，发现这个版本还未有该功能，因此看起来连main函数都没跑到。

后面的调试方法是在exit函数里添加打印调用栈功能。按《[FAQ15114]如何获取进程的native调用栈？》添加代码，结果M版本已不适用，无法编译成功。

为了快速定位问题，直接在exit里判断是vtservice后就直接abort，通过抓coredump分析。

修改代码如下：

char \*strstr(const char \*s, const char \*find);  
  
void exit(int status)  
{  
   
 /\* add this block \*/  
 if (strstr(getprogname(), "vtservice") && status == 1)  
 {  
 abort();  
 }  
/\* add end \*/

复现抓取了ne db，用GAT解开db，并结合对应的symbols文件（symbols目录里的文件必须和db一致），利用工具E-Consulter分析，分析报告如下：

== 异常报告v2.9(仅供参考) ==  
报告解读: MediaTek On-Line> Quick Start> E-Consulter之NE/KE分析报告解读> NE分析报告  
详细描述: 从错误的地址(0x00000000)读数据, 请结合崩溃线程调用栈检查相关代码  
版本 : alps-mp-m0.mp9/eng build  
发生时间: Fri Jan 1 00:00:00 CST 2016  
命令行 : /system/bin/vtservice  
进程标识符(pid): 4051, 父进程标识符(ppid): 1 (/init)  
进程状态: 正在运行  
优先级 : 120 (0~99: 实时进程, 100~139: 普通进程)  
  
  
== 线程信息(共1个线程) ==  
当前线程信息:  
 线程名: /system/bin/vtservice, 暂停, 线程标识符(tid): 4051 (主线程)  
 本地调用栈:  
 linker \_\_dl\_strstr(s=0, find=0xF728CA4D) + 30 <bionic/libc/upstream-openbsd/lib/libc/string/strstr.c:49>  
 linker \_\_dl\_exit(参数1=1) + 24 <bionic/libc/stdlib/exit.c:57>  
 linker \_\_linker\_init\_post\_relocation() + 520 <bionic/linker/linker.cpp:3219>  
 linker \_\_dl\_\_\_linker\_init(raw\_args=0xFFB4D890) + 830 <bionic/linker/linker.cpp:3426>  
 linker \_start() + 4 <bionic/linker/arch/arm/begin.S:33>  
 == 栈结束 ==  
 对应汇编指令:  
 行号 地址 指令 提示  
 bionic/libc/upstream-openbsd/lib/libc/string/strstr.c  
 41 : F727E876: MOV R5, R0 ; \_\_dl\_strstr()参数 1可能有问题  
 49 : F727E88E: MOV R6, R5  
 F727E890: LDRB R3, [R6], #0x1! ; 线程停止在这里  
 当时的寄存器值:  
 R0: 00000008, R1: F728CA58, R2: 63697672, R3: 00000000, R4: F728CA4D, R5: 00000000, R6: 00000000, R7: FFB4D628  
 R8: 00000076, R9: 00000008, R10: FFB4D6AC, R11: FFB4D64C, R12: 80808000, SP: FFB4D628, R14: F727E88D, PC: F727E891  
  
  
== C堆检查 ==  
分配器: jemalloc, 最多允许使用: 4GB, 最多使用: 0B, 当前使用: 0B, 泄露阈值: 128MB, 调试等级: 0  
  
正常  
  
  
== 日志信息 ==  
main log:  
01-01 00:00:00.308 4051 4051 F libc : Fatal signal 11 (SIGSEGV), code 1, fault addr 0x0 in tid 4051 (vtservice)  
  
  
分析NE还需以下文件, 请提供out/target/product/$proj/symbols目录下相应的文件(必须同一次编译生成, 如果是lib则选择/system/lib目录下的文件):  
libudf.so  
vtservice

问题出在linker，先查看对应的代码：

3216 ElfW(Ehdr)\* elf\_hdr = reinterpret\_cast<ElfW(Ehdr)\*>(si->base);  
3217 if (elf\_hdr->e\_type != ET\_DYN) {  
3218 \_\_libc\_format\_fd(2, "error: only position independent executables (PIE) are supported.\n");  
3219 exit(EXIT\_FAILURE);  
3220 }

看起来vtservice不是PIE引起的exit(1)，马上查看vtservice，通过readelf看是否是DYN，命令如下：

* arm-linux-android-readelf -aW vtservice

看到是DYN类型的，那就是PIE了，为何在这里报错，怀疑和：

* [MediaTek On-Line](https://online.mediatek.com/)> [Quick Start](https://online.mediatek.com/_layouts/15/mol/topic/ext/TopicHome.aspx)> 踩内存专题分析> native案例分析> TEE踩坏浮点寄存器引起SF NE

相关，结果确认了是tbase TEE，没有这个问题。

那是怎么回事？看到linker的log是直接输出到stderr的，那么直接在adb shell输入：

* #/system/bin/vtservice

就可以看到是什么错误信息了，结果看到：

CANNOT LINK EXECUTABLE: cannot locate symbol "\_ZN7android25MakeHEVCCodecSpecificDataEPKcPiS2\_" referenced by "/system/lib/libimsma.so"...

原来是符号找不到引起的link失败。为何之前会导向不是PIE错误呢，原因是编译器优化了，将所有exit(1)合并在一起，无从知道是哪里跳过来的了。

通过内部查找，发现这个符号是在libcomutils.so，检查出问题的版本的libcomutils.so，用readelf查看符号：

* arm-linux-android-readelf -aW libcomutils.so

搜索\_ZN7android25MakeHEVCCodecSpecificDataEPKcPiS2\_，结果确实没找到。

经过确认发现某个patch开始就存在这个符号了，不可能缺失。下载对应的patch，用上面的方法查看，发现有这个符号的。

那么问题就明显了，是没有合入这个so导致的问题。合入这个patch后发现还是有其他符号找不到，需要彻查所有patch合入的情况。

根本原因：

patch没有全部合入引起vtservice链接失败。

解决方法：

拿到的patch务必全部合入。

结语：

需要对linker、elf熟悉。

#### 4.环境变量引起linker时寻找库错误"is 32-bit instead of 64-bit"

问题背景：

经常出现下面的NE

Revision: '0'  
ABI: 'arm64'  
pid: 9544, tid: 9544, name: ls >>> ls <<<  
signal 6 (SIGABRT), code -6 (SI\_TKILL), fault addr --------  
 x0 0000000000000000 x1 0000000000002548 x2 0000000000000006 x3 0000000000000008  
 x4 ffffffffffffffff x5 0000000000000000 x6 0000008080808080 x7 2c33351f656e1f63  
 x8 0000000000000083 x9 ffffffffffffffdf x10 0000000000000000 x11 0000000000000001  
 x12 0000000000000018 x13 0000000000000000 x14 0000000000000000 x15 0034645024a4b2f3  
 x16 0000000000000003 x17 2207397d3179729a x18 000000000000003a x19 0000007b96195b40  
 x20 0000000000000006 x21 0000007b96195a98 x22 0000000000000000 x23 0000000000000005  
 x24 0000000000000001 x25 0000007fd208ce10 x26 0000007fd208ce40 x27 0000007b960ce010  
 x28 0000007b960d7811 x29 0000007fd208cc80 x30 0000007b96156f60  
  
backtrace:  
 #00 pc 0000000000075b04 /system/bin/linker64 (\_\_dl\_tgkill+8)  
 #01 pc 0000000000074f5c /system/bin/linker64 (\_\_dl\_pthread\_kill+64)  
 #02 pc 000000000006864c /system/bin/linker64 (\_\_dl\_raise+24)  
 #03 pc 0000000000065fdc /system/bin/linker64 (\_\_dl\_abort+52)  
 #04 pc 00000000000681dc /system/bin/linker64 (\_\_dl\_\_\_libc\_fatal+104)  
 #05 pc 000000000000fe18 /system/bin/linker64 (\_\_dl\_\_ZL29\_\_linker\_init\_post\_relocationR19KernelArgumentBlocky+3716)  
 #06 pc 000000000000eef4 /system/bin/linker64 (\_\_dl\_\_\_linker\_init+528)  
 #07 pc 0000000000006cd0 /system/bin/linker64 (\_start+4)

12-24 00:00:03.878 9544 9544 F libc : CANNOT LINK EXECUTABLE "ls": "/system/vendor/lib/libdirect-coredump.so" is 32-bit instead of 64-bit

12-24 00:00:03.887 9544 9544 F libc : Fatal signal 6 (SIGABRT), code -6 in tid 9544 (ls)  
12-24 00:00:03.891 1773 1773 D AEE\_AED : $===AEE===AEE===AEE===$  
12-24 00:00:03.891 1773 1773 D AEE\_AED : p 2 poll events 1 revents 1

复现概率高

分析过程：

从main log看，

12-24 00:00:03.878 9544 9544 F libc : CANNOT LINK EXECUTABLE "ls": "/system/vendor/lib/libdirect-coredump.so" is 32-bit instead of 64-bit  
12-24 00:00:03.887 9544 9544 F libc : Fatal signal 6 (SIGABRT), code -6 in tid 9544 (ls)  
12-24 00:00:03.891 1773 1773 D AEE\_AED : $===AEE===AEE===AEE===$

是在链接时找错库了，在执行ls命令时，出现NE,开始怀疑是不是客户下面没有64bit的libdirect-coredump.so这个库文件。

让客户从手机中看到底有没有这个库，看能不能执行ls命令，客户的回答是肯定可以的。那就只好看linker的源码了，为什么会找错库。

linker的源码位于bionic/linker下面，

先看看出错的位置

bool ElfReader::VerifyElfHeader() {  
  
....  
 if (needed\_libraries\_count > 0 &&  
 !find\_libraries(&g\_default\_namespace, si, needed\_library\_names, needed\_libraries\_count,  
 nullptr, &g\_ld\_preloads, ld\_preloads\_count, RTLD\_GLOBAL, nullptr,  
 /\* add\_as\_children \*/ true)) {  
 \_\_libc\_fatal("CANNOT LINK EXECUTABLE \"%s\": %s", args.argv[0], linker\_get\_error\_buffer());  
 } else if (needed\_libraries\_count == 0) {  
........

再看一下函数的调用流程:

\_\_linker\_init\_post\_relocation()->find\_libraries()->find\_library\_internal()->load\_library()

->open\_library()

然后在open\_library()中看到下面一段代码.

 // Otherwise we try LD\_LIBRARY\_PATH first, and fall back to the default library path  
 int fd = open\_library\_on\_paths(zip\_archive\_cache, name, file\_offset, ns->get\_ld\_library\_paths(), realpath);  
 if (fd == -1 && needed\_by != nullptr) {  
 fd = open\_library\_on\_paths(zip\_archive\_cache, name, file\_offset, needed\_by->get\_dt\_runpath(), realpath);  
 // Check if the library is accessible  
 if (fd != -1 && !ns->is\_accessible(\*realpath)) {  
 fd = -1;  
 }  
 }

注释已经写得很清楚了，先去尝试LD\_LIBRARY\_PATH这个路径，再去试default library path.

我们怎么知道LD\_LIBRARY\_PATH这个值是多少了？

在db下面PROCESS\_ENVIRONMENT这个文件中有记录，进程出现NE时，PROCESS\_ENVIRONMENT这个文件会把当时的环境变量记录下来。

我们在PROCESS\_ENVIRONMENT中可以看到

LD\_LIBRARY\_PATH=/vendor/lib:/system/lib

所以找到32位的库而没有找64位的。

那为什么LD\_LIBRARY\_PATH这个值会是/vendor/lib:/system/lib? 那这个就是当前进程可以设置的。

我们再看看当时出现问题的进程。

u:r:untrusted\_app:s0:c512,c768 u0\_a192   9544  9537  1852   756   1  30    10    0     0      ls  
u:r:untrusted\_app:s0:c512,c768 u0\_a192   9537  32496 8088   2496  1  30    10    0     0     sh  
u:r:untrusted\_app:s0:c512,c768 u0\_a192   32496 645   1203800 99704 2  20    0     0     0     com.tencent.qqpim

此因可以断定是com.tencent.qqpim这个有问题。为了验证这个，就改了一个应用程序.

while(1) {  
 sleep(10);  
 printf("system ls start\n");  
 putenv("LD\_LIBRARY\_PATH=/vendor/lib:/system/lib");  
 printf("%s 2\n",getenv("LD\_LIBRARY\_PATH"));  
 system("/system/bin/sh -c ls");  
 sleep(50);  
 printf("system ls end\n");  
}

 运行一下，出现相同的NE db。

根本原因：

应用程序本身的原因,可能版本太旧。

解决方法：

拿掉旧的应用程序

结语：

需要对linker熟悉。

#### 库包库导致app崩溃

问题背景

在安装使用安徽移动APP过程中发现，该APP一连网，就会Force Close，用eng版本复现，抓到NE db，里面包含coredump。

分析过程

利用工具E-Consulter分析，产生分析报告如下：

== 异常报告v3.2(仅供参考) ==  
报告解读: MediaTek On-Line> Quick Start> E-Consulter之NE/KE分析报告解读> NE分析报告  
详细描述: 从错误的地址(0x00000000)读数据, 请结合崩溃线程调用栈检查相关代码  
版本 : alps-mp-n0.mp1/eng build  
发生时间: Mon Dec 18 11:02:45 CST 2017  
命令行 : com.sitech.ac  
pid : 10334, ppid: 345 (zygote)  
进程状态: 正在运行  
优先级 : 120 (0~99: 实时进程, 100~139: 普通进程)  
  
== 线程信息(共82个线程) ==  
当前线程信息:  
 线程名: main, 可中断睡眠, tid: 10334 (主线程)  
 errno: 2  
 线程状态: 在调用JNI方法  
 本地调用栈:  
 libutils.so android::SharedBuffer::acquire() <external/libcxx/include/atomic:930>  
 libutils.so android::allocFromUTF8() + 76 <system/core/libutils/String16.cpp:37>  
 libutils.so android::String16::String16(参数1=0xBECB557C) + 8 <system/core/libutils/String16.cpp:160>  
 libdatabase\_sqlcipher.so 0x90FBC51D(参数1=0xAB8993F0, 参数2=0xBECB55DC, 参数3=0, 参数4=6) + 98  
 base.odex net.sqlcipher.CursorWindow.getString\_native() + 92  
 /dev/ashmem/dalvik-jit-code-cache 0x95EE4433()  
 /dev/ashmem/dalvik-jit-code-cache 0x95EE4F4F(参数2=0x12E833C0, 参数3=6)  
 system@framework@boot-framework.oat android.database.CursorWrapper.getString(参数1=0x6FE03554) + 46   
 libart.so art\_quick\_invoke\_stub\_internal() + 64  
......  
 对应汇编指令:  
 行号 地址 指令 提示  
 external/libcxx/include/atomic  
 930 : 92275462: LDXR R1, [R0] ; android::SharedBuffer::acquire()参数 1可能有问题  
 当时的寄存器值:  
 R0: 00000000, R1: 00000000, R2: 00000000, R3: 8F700000, R4: 00000000, R5: 8F700137, R6: AC0D5008, R7: 904B8AB8  
 R8: 12E21678, R9: AB884400, R10: 00000000, R11: BECB571C, R12: 92282C4C, SP: BECB5550, R14: 9227681D, PC: 92275463

可以看到android::SharedBuffer::acquire()参数1 = NULL导致了NE。参数1来自system/core/libutils/String16.cpp：

static inline char16\_t \*getEmptyString()  
{  
 gEmptyStringBuf->acquire(); /\* 这里调用了acquire(), 其中gEmptyStringBuf = NULL导致了NE \*/  
 return gEmptyString;  
}

我们发现gEmptyStringBuf是全局静态变量，是由initialize\_string16()初始化的：

void initialize\_string16()  
{  
 SharedBuffer\* buf = SharedBuffer::alloc(sizeof(char16\_t));  
 char16\_t\* str = (char16\_t\*)buf->data();  
 \*str = 0;  
 gEmptyStringBuf = buf;  
 gEmptyString = str;  
}

而initialize\_string16()是被如下函数调用：

class LibUtilsFirstStatics  
{  
public:  
 LibUtilsFirstStatics()  
 {  
 initialize\_string8();  
 initialize\_string16();  
 }  
 ~LibUtilsFirstStatics()  
 {  
 terminate\_string16();  
 terminate\_string8();  
 }  
};  
  
static LibUtilsFirstStatics gFirstStatics;

在LibUtilsFirstStatics的构造函数里调用了initialize\_string16()。由于定义了gFirstStatics全局静态对象，这个对象只有一个用处，那就是在库加载时调用构造函数而已。因此gEmptyStringBuf在libutils.so加载时初始化。

可是为何在这个进程里的gEmptyStringBuf = NULL呢？是否可能是：

* 库加载时没有调用LibUtilsFirstStatics()或initialize\_string16()
* 库卸载时调用了~LibUtilsFirstStatics()，释放了gEmptyStringBuf。
* gEmptyStringBuf被踩坏

我们在分析报告里搜索了link关键字，没有找到任何对应的函数，因此排除第2点。

而第1点，更不可能了，难道linker坏掉了？从第3点出发，用trace32加载debug.cmm，看下gEmptyStringBuf：

#### 

#### 居然有4个gEmptyStringBuf，不合理，后来发现1个libutils.so就有2个gEmptyStringBuf，1个是String8的，1个是String16的。那为何有4个？查看PROCESS\_MAPS：

#### 

发现libutils.so被加载了2次，这是为何？这是Android N及之后linker的一个新功能：namespace。namespace用于控制native lib的链接，避免app随意使用私有native API。如果用不好就会导致不同的namespace可能加载同样的库。

好了，回到正题，即使加载了2次，4个gEmptyStringBuf也应该被初始化，但打开变量发现都没被初始化，这就诡异了。

难道是linker出问题了（怀疑到第1点）？其实这是linker的链接规则引起的问题，有些库利用这个规则hook。

比如libudf.so，里面同样定义了和libc.so的malloc/free等函数，程序在调用malloc函数时，不是调用了libc.so的malloc，而是转调用到libudf.so的malloc，达到hook目的，用于debug malloc。

于是我们找下是否有人定义同样的函数initialize\_string16()：

#### 

#### 我们发现除了2个libutils.so之外还有一处定义了initialize\_string16():

#### 

查看0x922DF630是属于哪个lib：



终于找到凶手了，是libsqlcipher\_android.so。libsqlcipher\_android.so为何定义initialize\_string16()？

怀疑libsqlcipher\_android.so静态连接了libutils.so，部分symbol被export出来，扰乱了链接顺序。

原本应该是：

1. 0x92268000地址处的libutils.so加载时调用自身的initialize\_string16()函数初始化自身的gEmptyStringBuf
2. 0xac418000地址处的libutils.so加载时调用自身的initialize\_string16()函数初始化自身的gEmptyStringBuf

但由于libsqlcipher\_android.so也定义了initialize\_string16()，导致变成了：

1. 0x92268000地址处的libutils.so加载时调用libsqlcipher\_android.so的initialize\_string16()函数初始化libsqlcipher\_android.so的gEmptyStringBuf
2. 0xac418000地址处的libutils.so加载时调用libsqlcipher\_android.so的initialize\_string16()函数初始化libsqlcipher\_android.so的gEmptyStringBuf
3. 0x92268000地址处的libutils.so里的gEmptyStringBuf和0xac418000地址处的libutils.so里的gEmptyStringBuf都没有初始化

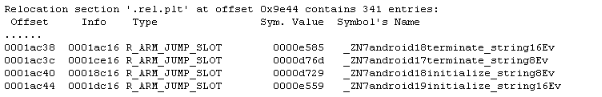
额外的问题是不应该加载2次libutils.so，应该是app本身使用里私有化native API。

根本原因

有2个原因：

* 应用本身携带的库有问题，扰乱链接。
* libutils.so被Mediatek修改过，导致initialize\_string16()被export出去，如果没有export出去就不会有问题了。patch：ALPS02995763，将还原为AOSP的做法。如下是AOSP的libutils.so（左边）和有问题的libutils.so（右边）的initialize\_string16()符号的属性，DEFAULT将被export出去：

通过.rel.plt可以明确看到有问题的libutils.so的initialize\_string8等需要链接，而AOSP的libutils.so则不需要！



解决方法

请第3方修复或者patch ALPS02995763

结语

需要熟悉linker，熟悉namespace。

#### ART NE

问题背景：

执行开关机测试，发生多笔system server NE

分析过程：

通过工具解析的分析报告如下：

报告解读: MediaTek On-Line> Quick Start> E-Consulter之NE/KE分析报告解读> NE分析报告  
详细描述: 从错误的地址(0xFFFFFFFFFFFFFFE8)读数据, 请结合崩溃线程调用栈检查相关代码  
版本 : alps-mp-o1.mp7/user build  
发生时间: Sun Jun 24 07:31:16 CST 2018  
命令行 : system\_server  
pid : 838, ppid: 500 (zygote64)  
进程状态: 不可中断睡眠  
优先级 : 118 (0~99: 实时进程, 100~139: 普通进程)

本地调用栈:  
libart.so art::DumpKernelStack(参数1=0x0000007612CCC2B0, 参数2=2430, 参数3=0x000000761B550F2F, 参数4=0) + 160 <external/libcxx/include/string:1226>  
libart.so art::DumpUnattachedThread(os=0x0000007612CCC2B0) + 340 <art/runtime/thread\_list.cc:164>  
libart.so art::ThreadList::DumpUnattachedThreads(this=0x000000761BB09000, os=0x0000007612CCC2B0, dump\_native\_stack=false) + 612 <art/runtime/thread\_list.cc:189>  
libart.so art::ThreadList::DumpForSigQuit(this=0x000000761BB09000, os=0x0000007612CCC2B0) + 928 <art/runtime/thread\_list.cc:156>  
libart.so art::Runtime::DumpForSigQuit(this=0x000000761BABD600, os=0x0000007612CCC2B0) + 196 <art/runtime/runtime.cc:1677>  
libart.so art::SignalCatcher::HandleSigQuit(this=0x0000007612839960) + 1936 <art/runtime/signal\_catcher.cc:194>  
libart.so art::SignalCatcher::Run(arg=0x0000007612839960) + 348 <art/runtime/signal\_catcher.cc:271>  
libc.so 0x000000769C88C0E4(参数1=0x0000007612CCC4F0) + 36  
libc.so 0x000000769C842DB8() + 68  
== 栈结束 ==

NE在art/runtime/native\_stack\_dump.cc:388

kernel\_stack\_frames.pop\_back(); //访问0xFFFFFFFFFFFFFFE8这个错误的地址

查看kernel\_stack\_filename结构体：

kernel\_stack\_filename = (

    \_\_r\_ = (

      std::\_\_1::\_\_libcpp\_compressed\_pair\_imp<std::\_\_1::basic\_string<char, std::\_\_1::char\_traits<char>, std::\_\_1::allocator<ch

        \_\_l = (\_\_cap\_ = 0x31, \_\_size\_ = 0x1A, \_\_data\_ = 0x00000078D8BC9D30 = end+0x1376D10 -> "/proc/self/task/2718/stack"),

        \_\_s = (\_\_size\_ = 0x31, \_\_lx = 0x31, \_\_data\_ = ""),

        \_\_r = (\_\_words = (0x31, 0x1A, 0x00000078D8BC9D30)))))

kernel\_stack = (

    \_\_r\_ = (

      std::\_\_1::\_\_libcpp\_compressed\_pair\_imp<std::\_\_1::basic\_string<char, std::\_\_1::char\_tra

        \_\_l = (\_\_cap\_ = 0x0, \_\_size\_ = 0x0, \_\_data\_ = 0x0 =  -> NULL),

        \_\_s = (\_\_size\_ = 0x0, \_\_lx = 0x0, \_\_data\_ = ""),

        \_\_r = (\_\_words = (0x0, 0x0, 0x0)))))

  kernel\_stack\_frames = (

    std::\_\_1::\_\_vector\_base<std::\_\_1::basic\_string<char, std::\_\_1::char\_traits<char>, std::\_\_1::allocator<char> >, std::\_\_1::

    std::\_\_1::\_\_vector\_base<std::\_\_1::basic\_string<char, std::\_\_1::char\_traits<char>, std::\_\_1::allocator<char> >, std::\_\_1::allocator<std::\_\_1::basic\_string<char, std::\_\_1::char\_traits<char>, std::\_\_1::allocator<char> > > >::\_\_end\_ = 0xFFFFFFFFFFFFFFE8

当前dump的thread是2718这个线程，而tid：2718在SYS\_PROCESSES\_AND\_THREADS 这个文件中已经找不到了

从bsp log看到在NE前1-2s，线程2718退出了：  
-2718 [002] .... 31.652599: sched\_process\_exit: comm=SharedPreferenc pid=2718 prio=120

所以ReadFileToString(kernel\_stack\_filename, &kernel\_stack)执行完kernel\_stack应该都是空的。

从以上分析推测，可能是线程当时恰好退出然后抓该线程的stack ，导致NE。为了验证此问题确实是因为2718线程退出导致，请客户将/art/runtime/native\_stack\_dump.cc的以下379，380，381，382这四行注释掉，然后发signal 3 给system server进程，结果触发同样的NE

371void DumpKernelStack(std::ostream& os, pid\_t tid, const char\* prefix, bool include\_count) {  
372 if (tid == GetTid()) {  
373 // There's no point showing that we're reading our stack out of /proc!  
374 return;  
375 }  
376  
377 std::string kernel\_stack\_filename(StringPrintf("/proc/self/task/%d/stack", tid));  
378 std::string kernel\_stack;  
379 if (!ReadFileToString(kernel\_stack\_filename, &kernel\_stack)) {  
380 os << prefix << "(couldn't read " << kernel\_stack\_filename << ")\n";  
381 return;  
382 }

根本原因：

检查ReadFileToString()这个function，下面红色这里返回true的本意是如果最后 read() return 0 的时候，就表示资料读完了，例如，read() 会依次 return 8K, 8K, 4K, 0，最后 read() return 0 的时候，这个function 就return 了，但是没有考虑到 read() 第一次就return 0的情况。

79 File file(file\_name, O\_RDONLY, false);  
80 if (!file.IsOpened()) {  
81 return false;  
82 }  
83  
84 std::vector<char> buf(8 \* KB);  
85 while (true) {  
86 int64\_t n = TEMP\_FAILURE\_RETRY(read(file.Fd(), &buf[0], buf.size()));  
87 if (n == -1) {  
88 return false;  
89 }  
90 if (n == 0) {  
91 return true;  
92 }  
93 result->append(&buf[0], n);  
94 }

 从code 來看，本来应该是假设File file(file\_name, O\_RDONLY, false); 有成功，就应该可以读到资料，  
但就在open -> read 的时间，thread 就exit 了

解决方法：

 if (n == 0) {

   if(0==result->size())

       return false;  
   return true;

 }

#### Pthread Key 填满问题分析

#### **问题背景：**

一家客户反馈, 常时间跑camera 的压力测试, cameraserver 可能crash.

分析过程:

1. 与客户确认问题状态, 客户一开始只提交了只抓到了一个main log, 可以看到cameraserver 因为NE 退出.  为了确认问题点, 请客户复现抓到了cameraserver 的coredump.  并且提供了一个不匹配的第三方symbols, 以及其他的symbols.

2. 从coredump 解析.coredump,  最后crash 是因为:

libc.so                 abort(参数4=key\_map + 1120) + 4 <bionic/libc/arch-arm/bionic/abort\_arm.S:43>  
libst\_personblur.so     emutls\_init() + 16   
libc.so                 pthread\_once() + 40  <bionic/libc/bionic/pthread\_once.cpp:71>  
libst\_personblur.so     \_\_emutls\_get\_address(obj=0xCCFC7780) +68

根据客户的symbols 利用函数匹配可以抓到对应的汇编:

#### 

#### 即call \_\_gthread\_key\_create => pthread\_key\_create 后, check 返回值错误, 然后主动call abort 重启.

#### 而pthread\_key\_create 失败的原因是, 因为pthread key 已经塞满了.  解析pthread 的key\_map 可以看到完整的pthread key 的情况, 特别是知道它的析构函数.

(seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCAB58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xD1158350),  
    (seq = 0x1, key\_destructor = 0xDDFD8350),  
    (seq = 0x1, key\_destructor = 0xD4558350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xD2358350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xD2358350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCFE4A350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCEDD8350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCE451350),  
    (seq = 0x1, key\_destructor = 0xCDACA350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCEDD8350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCAB58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350),  
    (seq = 0x1, key\_destructor = 0xCCC58350))  
      
    .....

    可以看到大面积的pthread key 都是同样的一个析构函数 0xCCC58350,  然后我们解析这个析构函数的汇编:  
cc5df000-ccdb4000 r-xp 00000000 103:02 4854                              /system/vendor/lib/libst\_personblur.so  
ccdb4000-ccdc7000 r--p 007d4000 103:02 4854                              /system/vendor/lib/libst\_personblur.so  
ccdc7000-ccfcf000 rw-p 007e7000 103:02 4854                              /system/vendor/lib/libst\_personblur.so

 可以看到:  
           NUD:CCC58350 |E92D40F8            push    {r3-r7,r14}  
           NUD:CCC58354 |E5907000            ldr     r7,[r0]  
           NUD:CCC58358 |E1A04000            cpy     r4,r0  
           NUD:CCC5835C |E1A06000            cpy     r6,r0  
           NUD:CCC58360 |E3A05000            mov     r5,#0x0          ; r5,#0  
           NUD:CCC58364 |E1550007            cmp     r5,r7  
           NUD:CCC58368 |0A000006            beq     0xCCC58388  
           NUD:CCC5836C |E5B63004            ldr     r3,[r6,#0x4]!  
           NUD:CCC58370 |E3530000            cmp     r3,#0x0          ; r3,#0  
           NUD:CCC58374 |0A000001            beq     0xCCC58380  
           NUD:CCC58378 |E5130004            ldr     r0,[r3,#0xFFFFFFFC]  
           NUD:CCC5837C |EBE6B4D1            bl      0xCC6056C8       ; free  
           NUD:CCC58380 |E2855001            add     r5,r5,#0x1       ; r5,r5,#1  
           NUD:CCC58384 |EAFFFFF6            b       0xCCC58364  
           NUD:CCC58388 |E1A00004            cpy     r0,r4  
           NUD:CCC5838C |E8BD40F8            pop     {r3-r7,r14}  
           NUD:CCC58390 |EAE6B4CC            b       0xCC6056C8       ; free  
           NUD:CCC58394 |EAE6B645            b       0xCC605CB0       ; pthread\_getspecific  
           NUD:CCC58398 |EAE6B5CF            b       0xCC605ADC       ; pthread\_setspecific  
           NUD:CCC5839C |E92D4070            push    {r4-r6,r14}  
           NUD:CCC583A0 |E5905004            ldr     r5,[r0,#0x4]  
           NUD:CCC583A4 |E1A06000            cpy     r6,r0  
           NUD:CCC583A8 |E3550004            cmp     r5,#0x4          ; r5,#4  
           NUD:CCC583AC |E5900000            ldr     r0,[r0]  
           NUD:CCC583B0 |8A000006            bhi     0xCCC583D0  
           NUD:CCC583B4 |E2800004            add     r0,r0,#0x4       ; r0,r0,#4  
           NUD:CCC583B8 |EBE6B4BC            bl      0xCC6056B0       ; malloc  
           NUD:CCC583BC |E3500000            cmp     r0,#0x0          ; r0,#0  
           NUD:CCC583C0 |15800000            strne   r0,[r0]  
           NUD:CCC583C4 |12804004            addne   r4,r0,#0x4       ; r4,r0,#4  
           NUD:CCC583C8 |1A00000A            bne     0xCCC583F8

再次利用函数匹配的方式:

            ZSR:0067CF00|E92D40F8  emutls\_destroy:                      push    {r3-r7,r14}  
                       |  
                     69|  
           ZSR:0067CF04|E5907000                                       ldr     r7,[r0]  
   
                       |  
                     67|  
           ZSR:0067CF08|E1A04000                                       cpy     r4,r0            ; r4  
           ZSR:0067CF0C|E1A06000                                       cpy     r6,r0            ; r6  
                       |  
                       |  
                     72|  
           ZSR:0067CF10|E3A05000                                       mov     r5,#0x0          ; r5  
           ZSR:0067CF14|E1550007                                       cmp     r5,r7            ; i,  
           ZSR:0067CF18|0A000006                                       beq     0x67CF38  
                       |  
                     74|  
           ZSR:0067CF1C|E5B63004                                       ldr     r3,[r6,#0x4]!  
           ZSR:0067CF20|E3530000                                       cmp     r3,#0x0          ; r3  
           ZSR:0067CF24|0A000001                                       beq     0x67CF30

Root cause:

即这个问题的原因应当是, vendor 在线程HPTD\_DQ\_DIP1启动的时候, 通过emutls\_init call 了 \_\_gthread\_key\_create ==> pthread\_key\_create, 但在线程退出的时候忘记删除key 了, 这个导致pthread key 累计越来越多, 最后爆掉了, 而对应key 的析构函数是 emutls\_destroy. 麻烦vendor 修正.

#### 错误munmap别人的内存引起NE

#### 问题背景

MTBF测试随机打中各种ART NE问题。

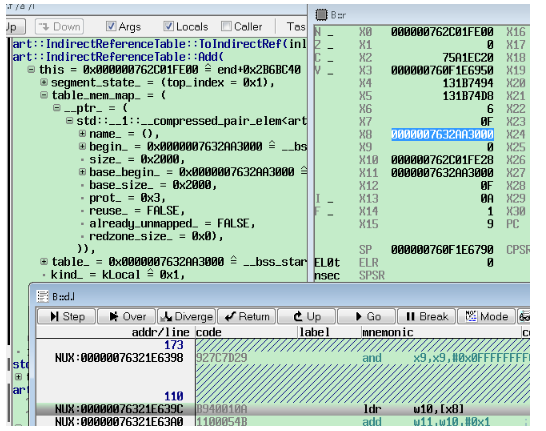
分析过程

分析几个db，利用工具SpOfflineDebugSuite分析，产生分析报告如下：

== 异常报告v2.0(仅供参考) ==  
报告解读: MediaTek On-Line> Quick Start> NE/KE分析报告解读> NE分析报告  
详细描述: 从错误的地址(0x0000007632AA3000)读数据, 请结合崩溃线程调用栈检查相关代码  
版本 : alps-mp-p0.mp1/userdebug build  
发生时间: Fri Oct 19 09:30:00 CST 2018  
命令行 : com.android.systemui  
pid : 8857, ppid: 12798 (zygote64)  
  
== 线程信息(共66个线程) ==  
当前线程信息:  
 线程名: Binder:8857\_B, 可中断睡眠, tid: 27728  
 errno: 22  
 线程状态: 正在运行  
 Java调用栈:  
 ...... java.lang.ref.Reference.getReferent() <本地方法>  
 boot.oat java.lang.ref.Reference.get()   
 boot.oat java.lang.ThreadLocal$ThreadLocalMap.set()   
 boot.oat java.lang.ThreadLocal.set()   
 boot-framework.oat android.os.Binder.execTransact()   
 == 栈结束 ==  
 本地调用栈:  
 libart.so art::IrtEntry::Add() + 12 <art/runtime/indirect\_reference\_table-inl.h:110>  
 libart.so art::IndirectReferenceTable::Add(this=0x000000762C01FE00, error\_msg=0x000000760F1E6950) + 1204 <art/runtime/indirect\_reference\_table.cc:313>  
 libart.so art::JNIEnvExt::AddLocalReference<\_jobject\*>(this=0x000000762C01FDE0, obj=0x000000760F1E69A4) + 60 <art/runtime/jni\_env\_ext-inl.h:29>  
 system@framework@boot.oat java.lang.ref.Reference.getReferent() + 124 <本地方法>  
 system@framework@boot.oat java.lang.ref.Reference.get(参数2=0x00000000131B74D8) + 40   
 system@framework@boot.oat java.lang.ThreadLocal$ThreadLocalMap.set(参数2=0x00000000131B7470, 参数3=0x0000000075A1EC20, 参数4=0) + 140   
 system@framework@boot.oat java.lang.ThreadLocal.set(参数2=0x0000000075A1EC20, 参数3=0) + 96   
 system@framework@boot-framework.oat android.os.Binder.execTransact() + 1260   
 libart.so art\_quick\_invoke\_stub(参数1=0x00000000717267D0, 参数2=0x000000760F1E6E60, 参数3=28, 参数4=0x000000762C088800, 参数5=0x000000760F1E6E40, 参数6=0x0000000074E0CC07) + 584 <art/runtime/arch/arm64/quick\_entrypoints\_arm64.S:1702>  
 libart.so art::ArtMethod::Invoke(this=0x00000000717267D0, self=0x000000762C088800, args=0x000000760F1E6E60, args\_size=28, result=0x000000760F1E6E40, shorty=0x0000000074E0CC07) + 200 <art/runtime/art\_method.cc:374>  
 libart.so art::(anonymous namespace)::InvokeWithArgArray(参数1=0x000000760F1E6F48, 参数2=0x00000000717267D0, 参数3=0x000000760F1E6E48, 参数4=0x000000760F1E6E40, 参数5=0x0000000074E0CC07) + 104 <art/runtime/reflection.cc:456>  
 libart.so art::InvokeVirtualOrInterfaceWithVarArgs(参数1=0x000000760F1E6F48, 参数2=10118, 参数3=0x00000000717267D0, 参数4=0x000000760F1E6F10) + 432 <art/runtime/reflection.cc:580>  
 libart.so art::JNI::CallBooleanMethodV(obj=10118, mid=0x00000000717267D0, args=0x000000760F1E70A0) + 648 <art/runtime/jni\_internal.cc:844>  
 libandroid\_runtime.so \_JNIEnv::CallBooleanMethod(this=0x000000762C01FDE0, obj=10118) + 120 <libnativehelper/include\_jni/jni.h:620>  
 libandroid\_runtime.so JavaBBinder::onTransact(this=0x0000007628A6BC40, code=1, data=0x000000760F1E7228, reply=0x000000760F1E71C0, flags=17) + 156 <frameworks/base/core/jni/android\_util\_Binder.cpp:344>  
 libbinder.so android::BBinder::transact(this=0x0000007628A6BC40, code=1, data=0x000000760F1E7228, reply=0x000000760F1E71C0, flags=17) + 136 <frameworks/native/libs/binder/Binder.cpp:129>  
 libbinder.so android::IPCThreadState::executeCommand(this=0x000000762C03A8C0, cmd=-2143260158) + 520 <frameworks/native/libs/binder/IPCThreadState.cpp:1121>  
 libbinder.so android::IPCThreadState::getAndExecuteCommand(this=0x000000762C03A8C0) + 156 <frameworks/native/libs/binder/IPCThreadState.cpp:458>  
 libbinder.so android::IPCThreadState::joinThreadPool(this=0x000000762C03A8C0, isMain=false) + 60 <frameworks/native/libs/binder/IPCThreadState.cpp:538>  
 libbinder.so android::PoolThread::threadLoop(this=0x0000007628A319E0) + 28 <frameworks/native/libs/binder/ProcessState.cpp:63>  
 libutils.so android::Thread::\_threadLoop(参数1=0x0000007628A319E0) + 280  
 libandroid\_runtime.so android::AndroidRuntime::javaThreadShell(args=0x0000007614FE2B60) + 140 <frameworks/base/core/jni/AndroidRuntime.cpp:1273>  
 libc.so \_\_pthread\_start(arg=0x000000760F1E74F0) + 36 <bionic/libc/bionic/pthread\_create.cpp:254>  
 libc.so \_\_start\_thread(fn=\_\_pthread\_start(), arg=0x000000760F1E74F0) + 68 <bionic/libc/bionic/clone.cpp:52>  
 == 栈结束 ==

挂在ART里，首先看下IndirectReferenceTable代码逻辑，不复杂。IndirectReferenceTable核心是一个数组，可以添加和删除obj。IndirectReferenceTable有3种类型：global，local，weakglobal。

其中global和weakglobal各只有一个IndirectReferenceTable实例，而local是per thread的。出问题的是某个thread的local IndirectReferenceTable内存异常。我们仔细看结构体：



访问0x7632AA3000时NE，看PROCESS\_MAPS:

7632aa2000-7632aa3000 rw-p 00010000 fc:00 3370 /system/lib64/libtombstoned\_client.so  
7632aa4000-7632aa5000 rw-p 00001000 00:04 27060917 /dev/ashmem/dalvik-indirect ref table (locals\_)

就没有0x7632AA3000对应的内存。但结构体base\_begin\_ = 0x7632AA3000。所以怀疑有人意外把IndirectReferenceTable mnumap掉。

后面测试又打中好机例，有些是在IndirectReferenceTable NE，有些是art其他内存被munmap引起的NE。

== 异常报告v3.2(仅供参考) ==  
报告解读: MediaTek On-Line> Quick Start> NE/KE分析报告解读> NE分析报告  
详细描述: 写数据0xA646F808到错误的地址(0xAB329000), 请结合崩溃线程调用栈检查相关代码  
版本 : alps-mp-p0.mp1.tc1sp/userdebug build  
发生时间: Thu Nov 8 04:22:59 CST 2018  
命令行 : com.android.launcher3  
pid : 7093, ppid: 6522 (zygote)  
  
== 线程信息(共38个线程) ==  
当前线程信息:  
 线程名: Jit thread pool worker thread 0, 可中断睡眠, tid: 7098  
 errno: 22  
 线程状态: 在调用JNI方法  
 本地调用栈:  
 libc.so memset() + 48 <bionic/libc/arch-arm/cortex-a7/bionic/memset.S:81>  
 libart-compiler.so art::HScheduler::Schedule(参数1=0xA3FFF1C0, 参数2=0xA646F790) + 550  
 libart-compiler.so art::HScheduler::Schedule(参数2=0xA608C000) + 202  
 libart-compiler.so art::HInstructionScheduling::Run() + 216  
 libart-compiler.so 0xA45089C1(参数1=0xA9FBDF80, 参数2=0xA608C000) + 90  
 libart-compiler.so art::OptimizingCompiler::RunOptimizations(参数1=0xA9FBDF80, 参数2=0xA608C000, 参数3=0xA608C100, 参数4=0xA3FFF738, 参数5=0xA3FFF4B0, 参数6=0xA3FFF638) + 506  
 libart-compiler.so art::OptimizingCompiler::TryCompile(参数1=0xA9FBDF80, 参数2=0xA3FFF688, 参数3=0xA3FFF660, 参数4=0xA3FFF648, 参数5=0xA3FFF738, 参数6=0xABD3C5C8, 参数7=0, 参数8=0xA3FFF638) + 1606  
 libart-compiler.so art::OptimizingCompiler::JitCompile(参数2=0xA4008000, 参数4=0xABD3C5C8, 参数5=0, 参数6=0) + 634  
 libart-compiler.so art::jit::JitCompiler::CompileMethod(参数2=0xABD3C5C8, 参数3=0xA4008000, 参数4=0) + 130  
 libart.so art::jit::Jit::CompileMethod(参数2=0xABD3C5C8, 参数3=0xA4008000, 参数4=1) + 440  
 libart.so 0xA9C49A9D(参数1=0x8EB769B0, 参数2=0xA4008000) + 422  
 libart.so art::ThreadPoolWorker::Run(参数1=0xAA00E9A0) + 44  
 libart.so art::ThreadPoolWorker::Callback(参数1=0xAA00E9A0, 参数2=art::ThreadPoolWorker::Callback()) + 94  
 libc.so \_\_pthread\_start(arg=0xA3FFF970) + 22 <bionic/libc/bionic/pthread\_create.cpp:254>  
 libc.so \_\_start\_thread(fn=\_\_pthread\_start()) + 24 <bionic/libc/bionic/clone.cpp:52>  
 == 栈结束 ==

0xAB329000对应的PROCESS\_MAPS：

ab328000-ab329000 rw-p 00000000 00:04 121878 /dev/ashmem/dalvik-CompilerMetadata (deleted)  
ab32a000-ab348000 rw-p 00002000 00:04 121878 /dev/ashmem/dalvik-CompilerMetadata (deleted)

dalvik-CompilerMetadata被munmap成2份了。

需要有个对策抓住凶手，观察到munmap的特征：

* munmap通常是一个page
* 基本都是落在ART申请的内存上
* 基本都存在：gralloc : Warning shared attribute region mapped at free. Unmapping这样的log

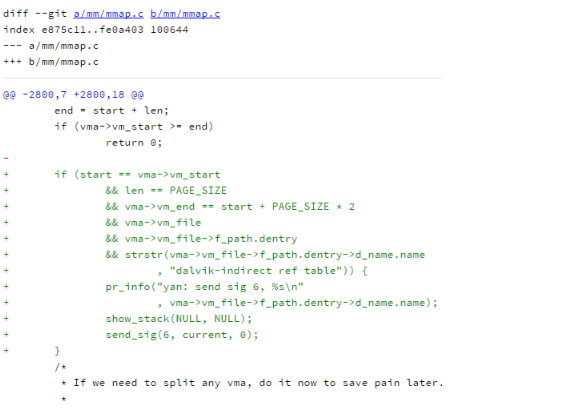
最后一点很重要，找到对应的代码：

if( hnd->attr\_base != MAP\_FAILED )  
{  
 ALOGW("Warning shared attribute region mapped at free. Unmapping");  
 munmap( hnd->attr\_base, PAGE\_SIZE );  
 hnd->attr\_base = MAP\_FAILED;  
}

刚好也是一个page，非常怀疑gralloc导致的NE。口说无凭，根据特征设计debugging方案：

* libc hook munmap函数
* kernel hook munmap函数

libc比较麻烦，最后考虑在kernel do\_munmap函数里加代码拦截：



经过几轮测试，终于有一例中枪，抓到db分析：

== 异常报告v3.2(仅供参考) ==  
报告解读: MediaTek On-Line> Quick Start> NE/KE分析报告解读> NE分析报告  
详细描述: 程序主动调用abort()  
版本 : alps-mp-p0.mp1.tc1sp/userdebug build  
发生时间: Thu Nov 8 04:57:08 CST 2018  
命令行 : com.android.systemui  
pid : 6938, ppid: 13254 (zygote)  
  
== 线程信息(共59个线程) ==  
当前线程信息:  
 线程名: FinalizerDaemon, 可中断睡眠, tid: 6947  
 errno: 11  
 线程状态: 在调用JNI方法  
 Java调用栈:  
 ...... android.graphics.GraphicBuffer.nDestroyGraphicBuffer() <本地方法>  
 boot-framework.oat android.graphics.GraphicBuffer.finalize()   
 boot-core-libart.oat java.lang.Daemons$FinalizerDaemon.doFinalize()   
 boot-core-libart.oat java.lang.Daemons$FinalizerDaemon.runInternal()   
 boot-core-libart.oat java.lang.Daemons$Daemon.run()   
 boot.oat java.lang.Thread.run()   
 == 栈结束 ==  
 本地调用栈:  
 libc.so munmap(参数1=0xA41AA000, 参数2=4096) + 12 <bionic/libc/arch-arm/syscalls/munmap.S:10>  
 gralloc.mt6750.so gralloc\_buffer\_attr\_free(hnd=0x9CF278A0) + 44 <vendor/mediatek/proprietary/hardware/gpu\_mali/mali\_midgard/r26p0-01rel0/product/android/gralloc/src/gralloc\_buffer\_priv.cpp:135>  
 gralloc.mt6750.so mali\_gralloc\_reference\_release(handle=0x9CF278A0, canFree=true) + 292 <vendor/mediatek/proprietary/hardware/gpu\_mali/mali\_midgard/r26p0-01rel0/product/android/gralloc/src/mali\_gralloc\_reference.cpp:144>  
 gralloc.mt6750.so mali\_gralloc\_release(device=0x86BD9E40, buffer=0x9CF278A0) + 6 <vendor/mediatek/proprietary/hardware/gpu\_mali/mali\_midgard/r26p0-01rel0/product/android/gralloc/src/mali\_gralloc\_public\_interface.cpp:269>  
 android.hardware.graphics.mapper@2.0-impl.so 0x9664FEF1(参数1=0x869AA480) + 10  
 android.hardware.graphics.mapper@2.0-impl.so 0x96650A35(参数1=0x89A9E510, 参数2=0x869C51D0) + 26  
 android.hardware.graphics.mapper@2.0.so 0xA666342D(参数1=0x89A9E510, 参数3=0x9CF278A0) + 84  
 libui.so android::Gralloc2::Mapper::freeBuffer(参数1=0x9AD2DB08, 参数2=0x9CF278A0) + 26  
 libui.so android::GraphicBufferMapper::freeBuffer(参数1=0x9AD2DAD8, 参数2=0) + 38  
 libui.so android::GraphicBuffer::free\_handle(参数1=0x9CF69B40) + 90  
 libui.so android::GraphicBuffer::~GraphicBuffer(参数1=0x9CF69B40) + 20  
 libutils.so android::RefBase::decStrong(this=0x9CF69B40, id=0x9CF2FE20) + 66 <system/core/libutils/RefBase.cpp:434>  
 libandroid\_runtime.so android::android\_graphics\_GraphicBuffer\_destroy(参数1=0x9CF20080, 参数2=0x89A9E58C, 参数3=0x9CF2FE20, 参数4=0) + 12 <frameworks/native/libs/ui/include/ui/ANativeObjectBase.h:47>  
 system@framework@boot-framework.oat android.graphics.GraphicBuffer.nDestroyGraphicBuffer(参数2=0x13649CE8) + 102 <本地方法>  
 system@framework@boot-framework.oat android.graphics.GraphicBuffer.finalize(参数2=0x13649CE8) + 60   
 system@framework@boot-core-libart.oat java.lang.Daemons$FinalizerDaemon.doFinalize(参数2=0x6F97DD80, 参数3=0x13427158) + 86   
 system@framework@boot-core-libart.oat java.lang.Daemons$FinalizerDaemon.runInternal(参数2=0x6F97DD80) + 466   
 system@framework@boot-core-libart.oat java.lang.Daemons$Daemon.run(参数2=0x6F97DD80) + 66   
 system@framework@boot.oat java.lang.Thread.run(参数1=0) + 64   
 libart.so 0xA2D48731() + 68  
 libart.so art\_quick\_invoke\_stub(参数1=0x6F9008A0, 参数2=0x89A9E87C, 参数3=4, 参数4=0xA26CAA00) + 224  
 libart.so art::ArtMethod::Invoke(参数1=0x6F9008A0, 参数2=0xA26CAA00, 参数3=0x89A9E87C, 参数4=4, 参数5=0x89A9E860, 参数6=0x70D7E975) + 136  
 libart.so 0xA2C82AB1(参数1=0x89A9E914, 参数2=0x6F9008A0, 参数3=0x89A9E86C, 参数4=0x89A9E860) + 52  
 libart.so art::InvokeVirtualOrInterfaceWithJValues(参数1=0x89A9E900, 参数2=0x89A9E914, 参数3=5, 参数4=0x6F9008A0, 参数5=0) + 320  
 libart.so art::Thread::CreateCallback(参数1=0xA26CAA00, 参数2=art::Thread::CreateCallback()) + 866  
 libc.so \_\_pthread\_start(arg=0x89A9E970) + 22 <bionic/libc/bionic/pthread\_create.cpp:254>  
 libc.so \_\_start\_thread(fn=\_\_pthread\_start()) + 24 <bionic/libc/bionic/clone.cpp:52>  
 == 栈结束 ==

同时kernel log有打印：

[39008.320416] (1)[6947:FinalizerDaemon]mmap: yan: send sig 6, dev/ashmem/dalvik-indirect ref table  
[39008.320434] (1)[6947:FinalizerDaemon]Backtrace:   
[39008.320465] (1)[6947:FinalizerDaemon][] (dump\_backtrace) from [] (show\_stack+0x18/0x1c)  
[39008.320483] (1)[6947:FinalizerDaemon] r6:a41aa000 r5:a41ab000 r4:d31f9588 r3:dc8cb077  
[39008.320519] (1)[6947:FinalizerDaemon][] (show\_stack) from [] (do\_munmap+0x43c/0x46c)  
[39008.320537] (1)[6947:FinalizerDaemon][] (do\_munmap) from [] (SyS\_munmap+0x48/0x5c)  
[39008.320545] (1)[6947:FinalizerDaemon] r10:00000800 r9:d6caa000 r8:c0207e64 r7:a41aa000  
[39008.320580] (1)[6947:FinalizerDaemon][] (SyS\_munmap) from [] (\_\_sys\_trace\_return+0x0/0x2c)  
[39008.320605] -(1)[6947:FinalizerDaemon][name:mtprof&][signal][6947:FinalizerDaemon] send death sig 6 to [6947:FinalizerDaemon:R]

和之前猜测gralloc是凶手吻合。

根本原因

gralloc错误将别人的memory给munmap掉

解决方法

修正gralloc代码

结语

Anrdoid非常庞大且更新快速，基本都是在解决问题过程中学习的，比如IndirectReferenceTable

#### 不预期收到sig 33导致NE

问题背景

P版本做reboot测试，概率性发生NE，陆续收到8台同样问题的手机，问题严重。

分析过程

利用工具SpOfflineDebugSuite分析，产生分析报告如下：

== 异常报告v2.1(仅供参考) ==  
报告解读: MediaTek On-Line> Quick Start> NE/KE分析报告解读> NE分析报告  
详细描述: 程序正常，有可能是硬件问题(检查PCB是否经过线路仿真)或外部触发的信号导致  
版本 : alps-mp-p0.mp3.tc19sp/user build  
发生时间: Fri Jan 4 08:45:14 CST 2019  
命令行 : system\_server  
pid : 1088, ppid: 560 (zygote64)  
  
== 线程信息(共179个线程) ==  
当前线程信息:  
 线程名: system\_server, 可中断睡眠, tid: 2896  
 本地调用栈:  
 libc.so \_exit\_with\_stack\_teardown() + 8 <bionic/libc/arch-arm64/bionic/\_exit\_with\_stack\_teardown.S:37>  
 libc.so pthread\_exit(return\_value=0) + 244 <bionic/libc/bionic/pthread\_exit.cpp:129>  
 == 栈结束 ==  
 对应汇编指令:  
 行号 地址 指令 提示  
 bionic/libc/arch-arm64/bionic/\_exit\_with\_stack\_teardown.S  
 37 : 0000007647F61A18: MOV X0, #0 ; 线程停止在这里  
 当时的寄存器值:  
 X0: 0000000000000000, X1: 0000000000106000, X2: 000000759E538468, X3: 0000000000000008  
 X4: 0000000000000000, X5: 00000075A7E3BCF0, X6: 0000000000000000, X7: 0000000100000002  
 X8: 00000000000000D7, X9: 1A18E1D44B69BD2C, X10: FFFFFFF87FFFFBFF, X11: 0000000000000001  
 X12: 0000000000000001, X13: 000000764803E050, X14: 00000000FFFFFFFF, X15: AAAAAAAAAAAAAAAB  
 X16: 0000007648032208, X17: 0000007647FB36B8, X18: 000000764803E000, X19: 000000759E5384F0  
 X20: 0000000000000000, X21: 000000759E5384F0, X22: 00000440000004E8, X23: 0000000000000004  
 X24: 000000759E538570, X25: 000000759E433000, X26: 00000075A3D78588, X27: 00000075C6A39438  
 X28: 00000075C6BEE048, X29: 000000759E538490, X30: 0000007647FC69E0, SP: 000000759E538460  
 PC: 0000007647F61A18

非常奇怪NE的位置并不会发生sig 11，而且执行到这里的时候stack已经munmap了。按下面思路分析：

1. 怀疑HW不良：不可能，项目刚开始规模测试，复现到的问题现象一致，都是NE。
2. 怀疑kernel有故意发送sig 11：和客户一起排查，没有查到相关的代码。
3. 搜索2896线程之前做过什么事情：查找SYS\_ANDROID\_LOG，发现如下log：

01-04 08:45:11.758 1088 2896 D VAPIBlackWhiteInfoServer: get Secure Config!  
01-04 08:45:11.759 1088 2896 E ActivityThread: Failed to find provider info for xxx  
01-04 08:45:11.759 1088 2896 D VAPIBlackWhiteInfoServer: cursor is null, lock failed, continue checking for update!  
01-04 08:45:11.759 1088 2896 W FeatureService: VAPI getSecureConfigToVAPITable failed! times:6

* + 搜索其他几个db也都是这样的log，直接找对应的代码，发现new thread之后执行一段代码然后就退出了，做的事情也很简单，没问题。

1. 在bsp log里搜索2896，发现关键点：
2. -1256 [002] .... 26.173165: sched\_fork\_time: comm=AudioService pid=1256 child\_comm=AudioService child\_pid=2896 fork\_time=52769 us  
    -1256 [002] .... 26.173542: sched\_fork\_time: comm=AudioService pid=1256 child\_comm=AudioService child\_pid=2897 fork\_time=49462 us  
    -1093 [000] d..1 26.199979: signal\_generate: sig=33 errno=0 code=-6 comm=Thread-30 pid=2896 grp=0 res=0  
    -2896 [002] d..1 26.200012: signal\_deliver: sig=33 errno=0 code=-6 sa\_handler=764c88b7e8 sa\_flags=18000004  
    -2896 [002] d..1 26.200058: signal\_generate: sig=11 errno=0 code=128 comm=Thread-30 pid=2896 grp=0 res=0  
    -2896 [002] d..1 26.200062: signal\_deliver: sig=11 errno=0 code=128 sa\_handler=55bddb2948 sa\_flags=18000004  
    -2896 [002] d..1 26.200103: signal\_generate: sig=11 errno=0 code=128 comm=Thread-30 pid=2896 grp=0 res=0  
    -2896 [002] d..1 26.200105: signal\_deliver: sig=11 errno=0 code=128 sa\_handler=0 sa\_flags=18000004

发现sig 11前有sig 33送过来，而sig 33是用来dump backtrace的，具体看libbacktrace代码。

回到coredump，sig 33一般是Signal Catcher发出

线程名: Signal Catcher, 不可中断睡眠, tid: 1093  
 线程状态: 在调用JNI方法  
 本地调用栈:  
 libc.so syscall(参数1=98, 参数2=0x000000759E64945C, 参数3=137, 参数4=2, 参数5=0x00000075C08AEF18, 参数6=0, 参数7=0x00000000FFFFFFFF) + 28 <bionic/libc/arch-arm64/bionic/syscall.S:41>  
 libc.so \_\_futex(ftx=0x000000759E64945C, value=2, timeout=0x00000075C08AEF18) + 40 <bionic/libc/private/bionic\_futex.h:45>  
 libc.so FutexWithTimeout() + 72 <bionic/libc/bionic/bionic\_futex.cpp:58>  
 libc.so \_\_futex\_wait\_ex(ftx=0x000000759E64945C, value=2, use\_realtime\_clock=false, abs\_timeout=0x00000075C08AEF18) + 140 <bionic/libc/bionic/bionic\_futex.cpp:63>  
 libc.so \_\_pthread\_cond\_timedwait(cond=0x000000759E64945C, abs\_timeout\_or\_null=0x00000075C08AEF18) + 84 <bionic/libc/bionic/pthread\_cond.cpp:182>  
 libc.so pthread\_cond\_timedwait(cond\_interface=0x000000759E64945C, mutex=0x000000759E649434, abstime=0x00000075C08AEF18) + 120 <bionic/libc/bionic/pthread\_cond.cpp:209>  
 libbacktrace.so ThreadEntry::Wait(this=0x000000759E649400, value=1) + 112 <system/core/libbacktrace/ThreadEntry.cpp:107>  
 libbacktrace.so BacktraceCurrent::UnwindThread(this=0x00000075C04E65A0, num\_ignore\_frames=0) + 324 <system/core/libbacktrace/BacktraceCurrent.cpp:204>  
 libart.so art::DumpNativeStack(参数1=0x00000075C08AF2B0, 参数2=2896, 参数3=0, 参数4=0x00000075C6B43E8A, 参数5=0, 参数6=0, 参数7=1) + 220 <art/runtime/native\_stack\_dump.cc:305>  
 libart.so art::DumpUnattachedThread(os=0x00000075C08AF2B0) + 68 <art/runtime/thread\_list.cc:168>  
 libart.so art::ThreadList::DumpUnattachedThreads(this=0x00000075C6D22000, os=0x00000075C08AF2B0, dump\_native\_stack=true) + 340 <art/runtime/thread\_list.cc:191>  
 libart.so art::ThreadList::DumpForSigQuit(this=0x00000075C6D22000, os=0x00000075C08AF2B0) + 900 <art/runtime/thread\_list.cc:158>  
 libart.so art::Runtime::DumpForSigQuit(this=0x00000075C6C4B700, os=0x00000075C08AF2B0) + 188 <art/runtime/runtime.cc:1828>  
 libart.so art::SignalCatcher::HandleSigQuit(this=0x00000075BD0130A0) + 1372 <art/runtime/signal\_catcher.cc:196>  
 libart.so art::SignalCatcher::Run(arg=0x00000075BD0130A0) + 256 <art/runtime/signal\_catcher.cc:264>  
 libc.so \_\_pthread\_start(arg=0x00000075C08AF4F0) + 36 <bionic/libc/bionic/pthread\_create.cpp:254>  
 libc.so \_\_start\_thread(fn=\_\_pthread\_start(), arg=0x00000075C08AF4F0) + 68 <bionic/libc/bionic/clone.cpp:52>  
 == 栈结束 ==

明显看到DumpNativeStack正在dump 2896这个tid的stack，正等待sig 33响应呢。用trace32分析BacktraceCurrent::UnwindThread的this，发现还在等待2896 signal handler执行，也就是说2896的sig 33 handler还没被执行。

理论上pthread\_exit不应该响应任何sig，从代码上也是这样：

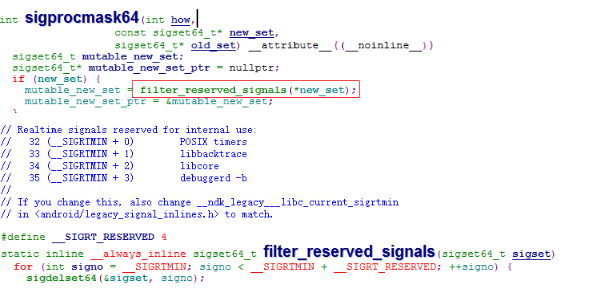
121   if (thread->mmap\_size != 0) {  
122   // We need to free mapped space for detached threads when they exit.  
123   // That's not something we can do in C.  
124    
125   // We don't want to take a signal after we've unmapped the stack.  
126   // That's one last thing we can do before dropping to assembler.  
127   ScopedSignalBlocker ssb;  
128   \_\_pthread\_unmap\_tls(thread);  
129   \_exit\_with\_stack\_teardown(thread->attr.stack\_base, thread->mmap\_size);  
130   }

ssb会阻止所有信号进来，但为何在\_exit\_with\_stack\_teardown还能收到sig 33呢？是否是kernel哪些地方会unmask block signal？检查代码也没发现有。

刚好db里有THREAD\_STATE，里面有：

SigQ: 1/14268  
SigPnd: 0000000000000000  
ShdPnd: 0000000000000000  
SigBlk: fffffff87ffbfaff

可以看到SigBlk里的bit33被清除了，还有其他bit被清除了。这就诡异了，再仔细检查代码看ScopedSignalBlocker，这个类包装了sigprocmask64，看sigprocmask64，终于发现问题：



从代码看会故意保留sig 32,33,34,35，这就是导致问题的根本原因。

解决方法

需要google fix。临时方案：

在文件头部申明：  
extern "C" int \_\_rt\_sigprocmask(int, const kernel\_sigset\_t\*, kernel\_sigset\_t\*, size\_t);  
然后  
// ScopedSignalBlocker ssb; // delete this line  
// add this block  
{  
   sigset64\_t new\_set, old\_set;  
  
   sigfillset64(&new\_set);  
   \_\_rt\_sigprocmask(SIG\_SETMASK, &new\_set, &old\_set, sizeof(new\_set));  
}

结语

从debugging中学习libbacktrace，linux signal机制。

遗留一个问题给大家思考：sig 33后面的sig 11从哪里来？是从page fault来吗？

#### camerahalserver NE -- 访问数组越界

#### **问题背景**

稳定性测试打出一例lib3a.flash.so NE

分析过程

1. camera owner根据如下log中的backtrace使用addr2line还原调用栈：

67081 02-02 23:34:03.415 17605 17605 F DEBUG   :

67082 02-02 23:34:03.415 17605 17605 F DEBUG   : backtrace:

67083 02-02 23:34:03.415 17605 17605 F DEBUG   :     #00 pc 0000000000010464  /vendor/lib64/lib3a.flash.so (NS3A::FlashAlgM::ForegroundSegment(double\*, int, double\*, double\*, double\*)+380)

67084 02-02 23:34:03.415 17605 17605 F DEBUG   :     #01 pc 000000000001105c  /vendor/lib64/lib3a.flash.so (NS3A::FlashAlgM::BuildFbTable()+356)

67085 02-02 23:34:03.415 17605 17605 F DEBUG   :     #02 pc 00000000000146a0  /vendor/lib64/lib3a.flash.so (NS3A::FlashAlgM::Estimate(NS3A::FlashAlgExpPara\*, NS3A::FlashAlgFacePos\*, int\*)+168)

67086 02-02 23:34:03.415 17605 17605 F DEBUG   :     #03 pc 00000000001bab74  /vendor/lib64/libcam.hal3a.v3.so (FlashMgrM::pfRun(FlashExePara\*, FlashExeRep\*)+476)

67087 02-02 23:34:03.415 17605 17605 F DEBUG   :     #04 pc 00000000001bb918  /vendor/lib64/libcam.hal3a.v3.so (FlashMgrM::doPfOneFrameNormal(FlashExePara\*, FlashExeRep\*)+632)

67088 02-02 23:34:03.415 17605 17605 F DEBUG   :     #05 pc 00000000001bb658  /vendor/lib64/libcam.hal3a.v3.so (FlashMgrM::doPfOneFrame(FlashExePara\*, FlashExeRep\*)+1256)

67089 02-02 23:34:03.415 17605 17605 F DEBUG   :     #06 pc 0000000000190c7c  /vendor/lib64/libcam.hal3a.v3.so (Task3AFlashBackImp::run(int, NS3Av3::TaskData const&)+2708)

67090 02-02 23:34:03.415 17605 17605 F DEBUG   :     #07 pc 0000000000195934  /vendor/lib64/libcam.hal3a.v3.so (TaskMgrImp::execute(NS3Av3::TASK\_UPDATE)+1148)

67091 02-02 23:34:03.415 17605 17605 F DEBUG   :     #08 pc 00000000000b17c0  /vendor/lib64/libcam.hal3a.v3.so (Hal3ARawImp::postCommand(NS3Av3::ECmd\_T, NS3Av3::ParamIspProfile\_T const\*)+3800)

67092 02-02 23:34:03.415 17605 17605 F DEBUG   :     #09 pc 00000000001f1204  /vendor/lib64/libcam.hal3a.v3.so (NS3Av3::Hal3AFlowCtrl::postCommand(NS3Av3::ECmd\_T, NS3Av3::ParamIspProfile\_T const\*)+1380)

//对应的代码如下：   
//从foregroundsegment分析并没有看到有特殊使用memory的地方，怀疑是别的地方踩坏了flash algo的memory

#### /对应的代码如下：  //从foregroundsegment分析并没有看到有特殊使用memory的地方，怀疑是别的地方踩坏了flash algo的memory

#### 

#### 2.请camera owner提供NE db，利用工具SpOfflineDebugSuite分析，产生分析报告如下：

报告解读: MediaTek On-Line> Quick Start> NE/KE分析报告解读> NE分析报告  
详细描述: 从错误的地址(0x0000007CC1A00000)读数据, 请结合崩溃线程调用栈检查相关代码  
版本 : alps-mp-p0.mp3/user build  
发生时间: Thu Jan 31 00:26:18 CST 2019  
命令行 : /vendor/bin/hw/camerahalserver  
pid : 645, ppid: 1 (init)  
  
== 线程信息(共157个线程) ==  
当前线程信息:  
 线程名: camerahalserver, 可中断睡眠, tid: 11297  
 本地调用栈:  
 lib3a.flash.so NS3A::FlashAlgM::ForegroundSegment(this=FlashAlgoMDev::getInstance(void)::singleton, fbCoef=0x0000007CC19FFD80, blks=80, outHr=0x0000007CC27640F0, outMr=0x0000007CC27640E8, outLr=0x0000007CC27640E0) + 380 <vendor/mediatek/proprietary/hardware/libcamera\_3a/libflash/mt6771/flash/FlashAlgM.cpp:2168>  
 lib3a.flash.so NS3A::FlashAlgM::BuildFbTable(this=FlashAlgoMDev::getInstance(void)::singleton) + 356 <vendor/mediatek/proprietary/hardware/libcamera\_3a/libflash/mt6771/flash/FlashAlgM.cpp:2558>  
 lib3a.flash.so NS3A::FlashAlgM::Estimate(this=FlashAlgoMDev::getInstance(void)::singleton, exp=0x0000007CD8921D38, pFaceInfo=0x0000007CC27646B8, isLowRef=0x0000007CD8921E64) + 168 <vendor/mediatek/proprietary/hardware/libcamera\_3a/libflash/mt6771/flash/FlashAlgM.cpp:3973>  
 libcam.hal3a.v3.so FlashMgrM::pfRun(参数1=0x0000007CD8921CF0, 参数2=0x0000007CC2769DC8, 参数3=0x0000007CC276DFC8) + 476  
 libcam.hal3a.v3.so FlashMgrM::doPfOneFrameNormal(参数1=0x0000007CD8921CF0, 参数2=0x0000007CC2769DC8, 参数3=0x0000007CC276DFC8) + 632  
 libcam.hal3a.v3.so FlashMgrM::doPfOneFrame() + 1256  
 libcam.hal3a.v3.so Task3AFlashBackImp::run(参数1=0x0000007CC230F260, 参数2=0, 参数3=0x0000007CC25F2338) + 2708  
 libcam.hal3a.v3.so TaskMgrImp::execute(参数1=0x0000007CC25F2280, 参数2=0) + 1148  
 libcam.hal3a.v3.so Hal3ARawImp::postCommand(参数1=0x0000007CD86E4D00, 参数2=8, 参数3=0x0000007CC2772F00) + 3800  
 libcam.hal3a.v3.so NS3Av3::Hal3AFlowCtrl::postCommand(参数1=0x0000007CD8933878, 参数2=8, 参数3=0x0000007CC27731E0) + 1380  
 libcam.hal3a.v3.so NS3Av3::Hal3AFlowCtrl::doUpdateCmd(参数1=0x0000007CD8933878, 参数2=0x0000007CC27731E0) + 660  
 libcam.hal3a.v3.so Thread3AImp::onThreadLoop(参数1=0x0000007CDE6281C0) + 1056  
 libcam.hal3a.v3.so std::\_\_1::\_\_thread\_proxy<\_\_1::tuple< <\_\_1::unique\_ptr<\_\_1::\_\_thread\_struct, \_\_1::default\_delete<?>>, (?)\*, Thread3AImp\*>>>(参数1=0x0000007CDE6DBE00) + 40  
 libc.so \_\_pthread\_start(arg=0x0000007CC27734F0) + 36 <bionic/libc/bionic/pthread\_create.cpp:254>  
 libc.so \_\_start\_thread(fn=\_\_pthread\_start(), arg=0x0000007CC27734F0) + 68 <bionic/libc/bionic/clone.cpp:52>  
 == 栈结束 ==  
 对应汇编指令:  
 行号 地址 指令 提示  
 vendor/mediatek/proprietary/hardware/libcamera\_3a/libflash/mt6771/flash/FlashAlgM.cpp  
 2168: 0000007CC65B5464: LDP D3, D4, [X11, #0x20] ; 线程停止在这里

#### vendor/mediatek/proprietary/hardware/libcamera\_3a/libflash/mt6771/flash/FlashAlgM.cpp:2168行在ForegroundSegment函数里，并且用trace32打开看到2168行的汇编代码应该是对x11附近的数据循环读取：

#### 

明显与上面的source code不对应

X11: 0000007CC19FFFE0，dump data看到这种明显是访问的地址超出了内存块size：

#### 

#### 请camera owner找到新版的source code，看起来代码是可以对应上了：

#### 

#### 很明显是对fbCoef数组访问越界，并非踩内存

根本原因

对fbCoef数组访问越界

解决方法

审查代码逻辑，对fbCoef数组正确操作

#### 11.lock不是原子导致了NE

问题背景

O版本做稳定性测试，极低概率性发生NE，问题持续了半年，问题严重。

分析过程

利用工具SpOfflineDebugSuite分析，产生分析报告如下：

== 异常报告v2.1(仅供参考) ==  
报告解读: MediaTek On-Line> Quick Start> NE/KE分析报告解读> NE分析报告  
详细描述: 从错误的地址(0x00000076CDA26574)读数据, 请结合崩溃线程调用栈检查相关代码  
版本 : alps-mp-o1.mp6/user build  
发生时间: Thu Jan 3 05:02:00 CST 2019  
命令行 : system\_server  
pid : 1039, ppid: 567 (zygote64)  
  
== 线程信息(共195个线程) ==  
当前线程信息:  
 线程名: main, 被跟踪, tid: 1039 (主线程)  
 线程状态: 在调用JNI方法  
 Java调用栈:  
 ...... java.lang.Thread.nativeCreate() <本地方法>  
 /system/framework/arm64/boot.vdex java.lang.Thread.start()   
 /system/framework/oat/arm64/services.vdex com.android.server.DropBoxManagerService$1.onReceive()   
 /system/framework/oat/arm64/services.vdex com.android.server.DropBoxManagerService$4.onChange()   
 /system/framework/arm64/boot-framework.vdex android.database.ContentObserver.onChange()   
 /system/framework/arm64/boot-framework.vdex android.database.ContentObserver.onChange()   
 /system/framework/arm64/boot-framework.vdex android.database.ContentObserver$NotificationRunnable.run()   
 /system/framework/arm64/boot-framework.vdex android.os.Handler.dispatchMessage()   
 /system/framework/arm64/boot-framework.vdex android.os.Looper.loop()   
 /system/framework/oat/arm64/services.vdex com.android.server.SystemServer.run()   
 /system/framework/oat/arm64/services.vdex com.android.server.SystemServer.main()   
 ...... java.lang.reflect.Method.invoke() <本地方法>  
 /system/framework/arm64/boot-framework.vdex com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run()   
 /system/framework/arm64/boot-framework.vdex com.android.internal.os.ZygoteInit.main()   
 == 栈结束 ==  
 本地调用栈:  
 libc.so Lock::unlock() + 20 <bionic/libc/private/bionic\_lock.h:74>  
 libc.so pthread\_create(thread\_out=0x0000007FC2B3ACB8, attr=0x0000007FC2B3ACF0, start\_routine=art::Thread::CreateCallback(), arg=0x00000076D79B9E00) + 872 <bionic/libc/bionic/pthread\_create.cpp:310>  
 libart.so art::Thread::CreateNativeThread(参数1=0x00000076F6678B20) + 608  
 libart.so 0x00000076F5FD7210(参数1=0x00000076F6678B20, 参数2=0x0000007FC2B3ADE4) + 156  
 system@framework@boot.oat java.lang.Thread.nativeCreate(参数2=0x0000000013BC1738, 参数3=0, 参数4=0) + 188 <本地方法>  
 system@framework@boot.oat java.lang.Thread.start(参数2=0x0000000013BC1738) + 148   
 services.odex com.android.server.DropBoxManagerService$1.onReceive(参数2=0x000000001451FEA0, 参数3=0x0000000012C40098, 参数4=0) + 184   
 services.odex com.android.server.DropBoxManagerService$4.onChange(参数2=0x00000000146E9260) + 140   
 system@framework@boot-framework.oat android.database.ContentObserver.onChange(参数2=0x00000000146E9260, 参数4=0x0000000013AC31C0) + 48   
 system@framework@boot-framework.oat android.database.ContentObserver.onChange(参数2=0x00000000146E9260, 参数4=0x0000000013AC31C0, 参数5=0) + 48   
 system@framework@boot-framework.oat android.database.ContentObserver$NotificationRunnable.run(参数2=0x0000000013BC0B50) + 84   
 system@framework@boot-framework.oat android.os.Handler.dispatchMessage(参数2=0x00000000146E9278, 参数3=0x0000000015B2E988) + 76   
 system@framework@boot-framework.oat android.os.Looper.loop() + 1452   
 services.odex com.android.server.SystemServer.run(参数2=0x0000000012C40130) + 3052   
 services.odex com.android.server.SystemServer.main() + 244   
 libart.so 0x00000076F613DFF0(参数1=0x0000000097F00E60, 参数2=0x0000007FC2B3B4D8, 参数3=4, 参数4=0x00000076F6660E00, 参数5=0x0000007FC2B3B4A8, 参数6=0x00000076DF45EB24) + 604  
 libart.so art::ArtMethod::Invoke(参数1=0x0000000097F00E60, 参数2=0x00000076F6660E00, 参数3=0x0000007FC2B3B4D8) + 264  
 libart.so 0x00000076F60619C4(参数1=0x0000007FC2B3B5B0, 参数2=0x0000000097F00E60, 参数3=0x0000007FC2B3B4C0, 参数4=0x0000007FC2B3B4A8, 参数5=0x00000076DF45EB24) + 100  
 libart.so art::InvokeMethod(参数1=0x0000007FC2B3B5B0, 参数5=1) + 1456  
 libart.so 0x00000076F5FE6CBC(参数1=0x00000076F6678B20, 参数2=0x0000007FC2B3B604) + 48  
 system@framework@boot.oat java.lang.reflect.Method.invoke(参数2=0x0000000012C40030, 参数3=0) + 180 <本地方法>  
 system@framework@boot-framework.oat com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(参数2=0x0000000012C401D0) + 132   
 system@framework@boot-framework.oat com.android.internal.os.ZygoteInit.main() + 2952   
 libart.so 0x00000076F613DFF0(参数1=0x00000000712D7240, 参数2=0x0000007FC2B3BA20, 参数3=4, 参数4=0x00000076F6660E00, 参数5=0x0000007FC2B3BA00, 参数6=0x00000076F4CBE5D9) + 604  
 libart.so art::ArtMethod::Invoke(参数1=0x00000000712D7240, 参数2=0x00000076F6660E00, 参数3=0x0000007FC2B3BA20) + 264  
 libart.so 0x00000076F60619C4(参数1=0x0000007FC2B3BB08, 参数2=0x00000000712D7240, 参数3=0x0000007FC2B3BA08, 参数4=0x0000007FC2B3BA00, 参数5=0x00000076F4CBE5D9) + 100  
 libart.so art::InvokeWithVarArgs(参数1=0x0000007FC2B3BB08, 参数2=0, 参数3=0x00000000712D7240, 参数4=0x0000007FC2B3BAD0) + 412  
 libart.so 0x00000076F5F67D20(参数4=0x0000007FC2B3BC60) + 612  
 libandroid\_runtime.so 0x0000007779F76E20(参数1=0x00000076F6678B20, 参数2=105, 参数4=17) + 120  
 libandroid\_runtime.so android::AndroidRuntime::start(参数1=0x0000007FC2B3BE68, 参数3=0x0000007FC2B3BE30, 参数4=0) + 964  
 app\_process64 main(argc=6, argv=0x0000007FC2B3CFC8) + 1328 <frameworks/base/cmds/app\_process/app\_main.cpp:0>  
 libc.so \_\_libc\_init(raw\_args=0x0000007FC2B3CFC0, onexit=0, slingshot=main(), structors=0x0000007FC2B3CFA8) + 88 <bionic/libc/bionic/libc\_init\_dynamic.cpp:124>  
 app\_process64 \_start\_main() + 80  
 == 栈结束 ==

对应代码如下：

int pthread\_create(pthread\_t\* thread\_out, pthread\_attr\_t const\* attr, void\* (\*start\_routine)(void\*), void\* arg) {  
 ErrnoRestorer errno\_restorer;  
  
 ......  
  
 // Publish the pthread\_t and unlock the mutex to let the new thread start running.  
 \*thread\_out = \_\_pthread\_internal\_add(thread);  
 thread->startup\_handshake\_lock.unlock(); /\* crash here \*/  
  
 return 0;  
}

#### 非常奇怪发现整个thread都不见了，用trace32看：

#### 

#### 难道还没unlock线程就退出了？看代码子线程必须要等父线程unlock才能往下跑。

static int \_\_pthread\_start(void\* arg) {  
 pthread\_internal\_t\* thread = reinterpret\_cast<pthread\_internal\_t\*>(arg);  
  
 // Wait for our creating thread to release us. This lets it have time to  
 // notify gdb about this thread before we start doing anything.  
 // This also provides the memory barrier needed to ensure that all memory  
 // accesses previously made by the creating thread are visible to us.  
 thread->startup\_handshake\_lock.lock();  
  
 \_\_init\_alternate\_signal\_stack(thread);  
  
 void\* result = thread->start\_routine(thread->start\_routine\_arg);  
 pthread\_exit(result);  
  
 return 0;  
}

如果不是子线程偷跑，那么又是谁把thread给munmap掉？如果是意外被munmap掉，那么存在如下问题：

* g\_thread\_list还残留有thread信息，但coredump来看g\_thread\_list正常，没有断裂，看起来子线程至少调用过\_\_pthread\_internal\_remove函数。

当时想了一个debugging方法，在thread结构体增加成员int isdone;在unlock之后才设置为1，在\_\_pthread\_internal\_remove()判断isdone是否为1，如果为0就abort。结果无法复现（原本概率极低）。

需要加大检查范围，查看bsp log：

#### 

#### 发现子线程果然自己退出去了，那么问题来了，还没unlock，子线程如何偷跑？是否是lock失效了？在仔细看汇编代码：

#### 

对应代码：

72 void unlock() {  
73   if (atomic\_exchange\_explicit(&state, Unlocked, memory\_order\_release) == LockedWithWaiter) {  
74   \_\_futex\_wake\_ex(&state, process\_shared, 1); /\* crash here \*/  
75   }  
76 }

一看，问题就明了了，在73行的时候已经unlock了，如果子线程被意外唤醒，那么是有可能先跑然后调用pthread\_exit()把thread释放，导致了lock也一起释放，那么在74行的process\_shared就无法访问，访问就会发生NE。

解决方法

google patch: https://android-review.googlesource.com/c/platform/bionic/+/937824

结语

深入汇编，仔细看现场。

利用bsp log观察流程。

#### 附录

#### 各种标准

#### **SUS** – (Single UNIX® Specification)     <http://www.unix.org/online.html> (注册一个账号即可下载) **LSB** – (Linux Standard Base: linux标准规范)     <http://refspecs.linuxbase.org/lsb.shtml> **DWARF** – (调试信息格式)     <http://dwarfstd.org/> **gABI** – (System V Application Binary Interface)     <http://refspecs.linuxbase.org/elf/gabi41.pdf> **C++ ABI**for Itanium     <http://refspecs.linuxbase.org/cxxabi-1.83.html> **ELF** – (Executable and Linkable Format)     <http://refspecs.linuxbase.org/elf/elf.pdf>