【团队分享】苍翼之刃：论File Descriptor泄漏如何导致Crash？

2015.5.7 腾讯Bugly 微信分享

这一期的团队分享，我们特邀苍翼之刃的开发负责人Jay，为大家分享在Android项目中遇到的一些Crash。

> 苍翼之刃
>
> 外文：BlazBlue Revolution Reburning，是由Arc Systerm Works 正式授权，并由 91Act 负责研发的横板动作手游，森利道全程监修，也是苍翼默示录系列唯一的一部横板动作手游。
>
> 游戏内部代号为：BBRR，全称是BLAZBLUE Revlution Reburning，续BBCT、BBCS、BBCS2、BBEX、BBCP 之后的又一力作。

背景

在*nix系统中，许多的资源都会被定义为File Descriptor（下面简称FD），例如普通文件、socket、std in/out/error等等。每个*nix系统中，单个进程可以使用的FD数量是有上限的。不同的*nix系统中，这个上限各有区别，例如在Android里面这个上限被限制为1024。

案例分析

在实际的Android开发过程中，我们遇到了一些奇奇怪怪的Crash，通过sigaction再配合libcorkscrew以及一些第三方的Crash Reporter都捕获不到发生Crash的具体信息，十分头疼。
然后我们通过Bugly上报的Java的CallStack观察发现这些Crash发现了一些共同的信息：

```
android.opengl.GLSurfaceView$EglHelper.throwEglException(GLSurfaceView.java:1178)
android.opengl.GLSurfaceView$EglHelper.swap(GLSurfaceView.java:1136)
android.opengl.GLSurfaceView$GLThread.guardedRun(GLSurfaceView.java:1463)
android.opengl.GLSurfaceView$GLThread.run(GLSurfaceView.java:1216)
```


看来是和OpenGL有关系，于是我们进一步对程序输出的log进行观察，又发现：

出错线程 | 其他线程 | 系统日志

```
1 E/Surface( 14388): dequeueBuffer: IGraphicBufferProducer::requestBuffer failed: -2147483646
2 W/Adreno-EGLSUB( 14388): <DequeueBuffer:720>: dequeue native buffer fail: Unknown error 2147483646, buffer=0x0, handle=0x0
3 W/Adreno-EGL( 14388): <qeglDrvAPI_eglSwapBuffers:3702>: EGL_BAD_SURFACE
4 W/GLThread( 14388): eglSwapBuffers failed: EGL_BAD_SURFACE
5 W/OpenGLRenderer( 14388): swapBuffers encountered EGL_BAD_SURFACE on 0xaf43d340, halting rendering...
```


从这个log里面我们获得了几个信息：

几乎所有出现这种Crash的设备，都是Adreno的GPU
几乎所有Crash都会伴随着requestBuffer failed

我们对我们已有的设备反复试验，确实了只有Adreno的设备（小米3，HTC M8，华为P7等）会在特定条件下出现这种奇奇怪怪的随机Crash。而其他设备例如小米Pad（Tegra），三星S3（Mali）等都不会出现这种问题。
这个问题确实头疼，在网上搜索了很久也没找到有用的信息。直到在某次小米3上再次测试的时候，发现了log里面还有一条必然出现的信息：

```
E/MemoryHeapBase(1870): error creating ashmem region: Too many open files
```

这个信息问接的指出了问题，也给了我们一些提示：似乎打开了过多的文件。于是靠着这个灵光，我们尝试着在程序中输出所有已打开的文件：

```
SHOW FILE HANDLES:
 0  (socket:[285038]): read-write
 1  (/dev/null): read-write
 2  (/dev/null): read-write
 3  (/dev/log/main): cloexec write-only
 4  (/dev/log/radio): cloexec write-only
 5  (/dev/log/events): cloexec write-only
 6  (/dev/log/system): cloexec write-only
 7  (/sys/kernel/debug/tracing/trace_marker): write-only
 8  (/dev/__properties__):
 9  (/dev/binder): cloexec read-write
10  (/dev/log/main): cloexec write-only
11  (/dev/log/radio): cloexec write-only
12  (/dev/log/events): cloexec write-only
13  (/dev/log/system): cloexec write-only
14  (/system/framework/framework-res.apk):
15  (/system/framework/core-libart.jar):
16  (pipe:[282578]): nonblock
17  (/dev/alarm):
18  (/dev/cpuctl/tasks): cloexec write-only
19  (/dev/cpuctl/bg_non_interactive/tasks): cloexec write-only
20  (socket:[282569]): read-write
21  (pipe:[282570]):
22  (pipe:[282570]): write-only
23  (pipe:[282578]): nonblock write-only
24  (anon_inode:[eventpoll]): read-write
25  (/data/app/---app_name---/base.apk):
26  (/data/data/---app_name---/databases/bugly_db): cloexec read-write
27  (socket:[285047]): read-write
28  (anon_inode:mali-8938): cloexec
29  (socket:[282605]): nonblock read-write
30  (socket:[283605]): nonblock read-write
31  (/dev/null): read-write
32  (/dev/ump): read-write
33  (socket:[285045]): nonblock read-write
34  (/dev/null): read-write
35  (/dev/mali): read-write
36  (anon_inode:mali-8938): cloexec
37  (anon_inode:mali-8938): cloexec
38  (/data/app/---app_name---/base.apk):
39  (anon_inode:mali-8938): cloexec
40  (anon_inode:mali-8938): cloexec
41  (/dev/null): read-write
42  (/dev/null): read-write
43  (/data/app/---app_name---/base.apk):
44  (/dev/null): read-write
45  (anon_inode:mali-8938): cloexec
46  (/data/data/---app_name---/files/DefaultFont.ttf):
47  (/data/app/---app_name---/base.apk):
48  (anon_inode:sync_fence):
49  (/dev/null): read-write
50  (socket:[285060]): cloexec read-write
52  (anon_inode:mali-8938): cloexec
53  (anon_inode:mali-8938): cloexec
54  (/dev/null): read-write
55  (anon_inode:sync_fence):
56  (pipe:[284134]): write-only
58  (anon_inode:sync_fence):
62  (anon_inode:sync_fence):
63  (anon_inode:sync_fence):
```

通过不停测试程序，发现已打开的文件数量一直有增无减，而当这些被打开的文件数量接近1024的时候，上面的eglSwapBuffers必然出错。于是乎我们得出一个中间结论：

如果程序打开的文件数量过多，会导致OpenGL swap buffer失败！

这从字面上看着似乎有些扯淡，因为这两者总感觉没啥联系。这个问题只会出现在Adreno的GPU上面，于是我们猜想：

- Adreno的驱动在swap buffer的时候，需要申请新的FD，这个FD可能是某些硬件IO，具体不得而知；
- 如果程序中其他的各种FD使用过多接近上限，会导致Adreno的驱动申请不到必要的FD，因此导致swap buffer失败。

这样看起来似乎貌比较有道理了。虽然sawp buffer本身是不会Crash的，他并没有raise任何signal，只是简单的返回了一个错误的结果，但这会导致上层逻辑出现异常。这些异常在不同的设备上表现不一样：

- 有的设备会在Java层的eglSwapBuffers触发Java层的Exception导致Crash；
- 有的设备不会出现异常，但是会导致OpenGL停止工作（halt rendering），其表现结果就是程序卡住无响应；
- 有的设备可能什么都不会发生，但是如果你的交互触发了其他逻辑：比如按回退键弹出对话框，对话框也需要FD，但是获得不到，那么弹出对话框的逻辑将抛出异常，

于是这就有了各种奇奇怪怪的Crash。

解决方案

通过对代码的排查，我们发现在使用SoundPool处理音效的时候，确实存在FD泄露的情况：

```
1  private SoundPool m_soundPool;
2  public int loadSound(String path) {
3      int soundID = m_soundPool.load(getAssets().openFd(path), 0);
4      return soundID;
5  }
6  public unloadSound(int soundID) {
7      m_soundPool.unload(soundID);
8  }
```

虽然我们在不需要这些音效的时候，对其进行了卸载处理，但不知道是SoundPool类自身的缺陷，还是我们的使用不当。在实际测试中我们发现unload过后，在load中通过openFd打开的FD并没有被释放掉。
强制调用System.gc()在一些设备（例如小米3）上可以释放掉这部分FD，但是另一些设备（例如HTC M8）即使强制gc还无法卸载掉它们，于是便出现了FD泄露的情况。

最终我们自行对这些FD进行管理，并且在unload的时候手动调用这些FD的close方法：

```
1  private SoundPool m_soundPool;
2  private HashMap<Integer, AssetFileDescriptor> m_soundFdMap
3  public int loadSound(String path) {
4      AssetFileDescriptor fd = getAssets().openFd(path);
5      int soundID = m_soundPool.load(fd, 0);
6      m_soundFdMap.put(soundID, fd);
7      return soundID;
8  }
9  public unloadSound(int soundID) {
10     m_soundPool.unload(soundID);
11     m_soundFdMap.get(soundID).close();
12 }
```

这之后FD再无泄露的情况发生，之前的各种设备上面的各种奇奇怪怪的Crash都被处理好了。

小结

这个问题粗略说起来就是：因为播放了太多的音效，导致Adreno底层渲染失败，以至于上层逻辑各种失措，产生了很多奇奇怪怪的Crash。
准确的解释应该是：程序中的FD泄露如同内存泄露一样是同样需要得到关注的问题，FD的耗尽如同内存的耗尽一样会导致程序的各种异常情况发生，但是前者不如后者那么知名也不如后者容易被察觉。