全志科技
Allwinner Technology

Android 系统内存泄漏

排查指导文档

1.0

2017.06.11

文档履历

版本号	日期	制/修订人	内容描述
1.0	2017.06.11		

目录

1. 前言

系统内存泄漏问题一直以来都是软件开发工程师最为头疼的问题，当隐秘的内存泄漏一旦发生，将在长期老化实验中发生``蝴蝶效应"，当工程师迷惑于问题现场时，``真凶"却隐藏在灯光的阴影深处，诡笑着软件工程师的捉衿见肘和无从下手。这个时候，软件工程师总会把问题解决的希望寄托在系统级工程师身上。然而任何看似复杂的系统问题，总可以通过严谨科学的系统思路和工具方法进行破解，最终从扑朔迷离的``灾难现场"反向追查到问题根源，找到``真凶"！本文以 Android 系统作为问题分析模型，将系统内存泄露问题进行不同层面的拆解，通过介绍问题处理思路和断案分析工具，帮忙软件开发工程师独立自主的解决内存泄漏问题。

2. 问题模型

Android 系统自下而上分别是 Linux 内核 (Linux Kernel)，系统库 (Libraries)，SOC 厂家适配层（HAL），Android 运行时环境 (Android Runtime), 框架层 (Application Framework) 以及应用层 (Application)。Android 系统内存泄漏问题从大的系统层面划分如下，可以分为用户空间内存泄漏和内核态空间内存泄漏。本节重点介绍不同层面内存泄漏问题分类以及如何通过工具去判断问题范畴。

Android 用户空间内存泄漏类型如下:
- Application 应用 JAVA 内存泄漏
- Application 应用 NativeHeap 内存泄漏
- 系统 native 进程 heap 泄漏

Android 内核空间内存泄漏类型如下:
- 内核 slab 内存泄漏
- 内核私有 page 使用泄漏
- 内核 Ion 内存泄漏

3. 问题调试

本章节将重点针对第 2 节内存泄漏问题的现象，指导 Android 软件工程师、内核驱动开发工程师如何使用工具进行问题判断，确认问题方向后使用更有效的方法和工具，对问题进行调试直至最终查到问题根源。

3.1 Android 用户空间内存泄漏现象

Android 用户空间进程分为 Application 应用进程、应用服务进程、系统框架层重要服务进程和系统 Native 重要进程，用户态进程的内存消耗主要是堆栈 heap 和共享内存 ashmem。heap 内存的数量主要是通过内核 Anon 页面进行统计，ashmem 内存的数量主要通过 shmem 统计，H6 方案 Sdk 中集成了 cpu_monitor 负载内存诊断工具，再确认 Android 系统 debugfs 文件系统挂载成功后，运行命令：

```
adb shell mount debugfs -t debugfs /sys/kernel/debug
adb shell cpu_monitor -m 1000
```

工具运行后如下图，需要观察工具抓取的 Anon 计数、Shmem 计数和 Swap-Free 计数的变化

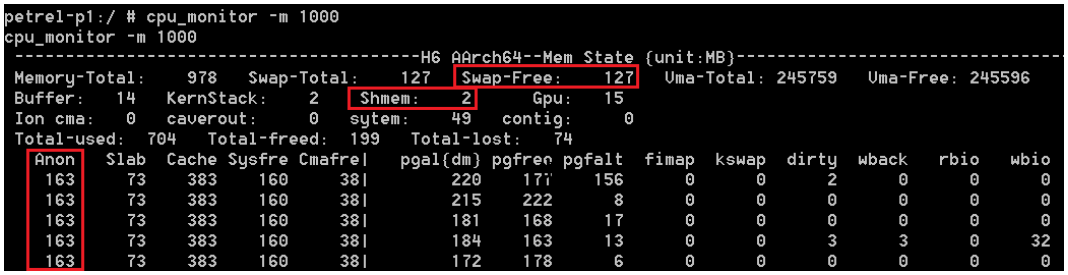


图 1: CpuMonitor 内存分布图

1.heap 内存泄漏现现象 Anon ↑ Swap-Free ↓如果用户空间出现 heap 泄漏，Anon 计数会随着系统运行时间的增长不断的累积上涨，同时 Swap-Free 计数会不断的缩减，因为 heap 的泄漏会造成旧的内存页面不断的被内核内存回收机制压缩到 Zram 的 Swap 空间，造成 Swap-Free 计数不断下降，如下图：

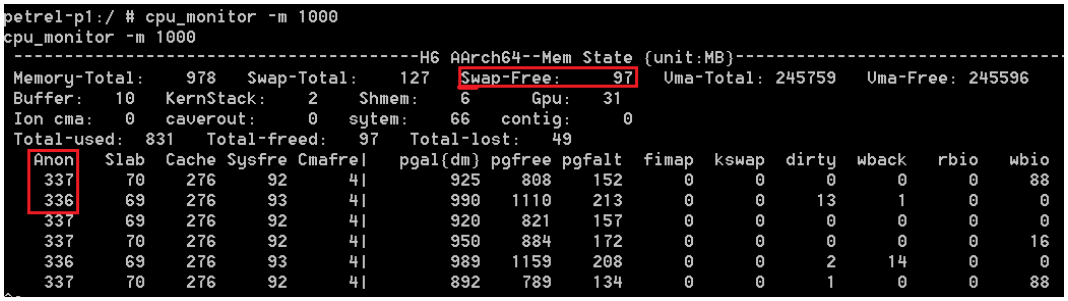


图 2: CpuMonitor 观察应用空间 HEAP 内存泄漏

2.Ashmem 内存泄漏现象 Shmem ↑ Swap-Free ↓如果用户空间出现 ashmem 内存泄漏，Shmem 计数会不断增加，同时当 shmem 累积到一定程度后，内核内存回收会将 shmem 内存页面压缩到 Zram 的 Swap 空间，造成 Swap-Free 计数不断下降，如下图：

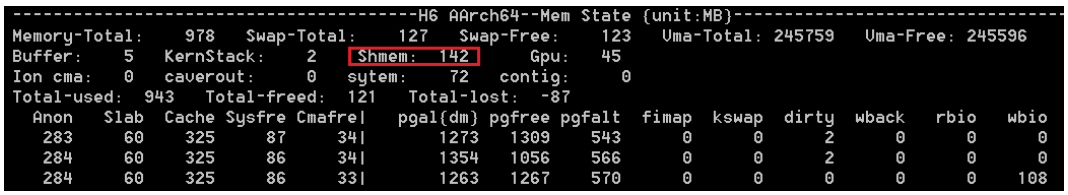


图 3: CpuMonitor 观察应用空间 ashmem 内存泄漏

通过 cpu_monitor 工具确认了系统用户空间存在内存泄漏后，就要通过更有效的工具和方法继续调试分析，以下小节中将重点针对问题的具体类型进行调试分析说明。

3.1.1 Application 应用 JAVA 内存泄漏

待补充……

3.1.2 Application 应用 NativeHeap 内存泄漏

待补充……

3.1.3 Android 系统 native 进程 heap 泄漏

待补充……

3.2 Android 内核空间内存泄漏现象

Android 内核态内存消耗，主要如下：- 内核基础内存系统预留内存，Slab 内存、文件系统 pagecache 内存、物理存储 block 层 buffer 内存、进程内核栈 KernelStack 内存、Zram 压缩内存，内核基础消耗的内存可以被追踪到。通过 cpu_monitor 工具可以抓取到内核基础内存如下图中所示：

Swap-Total - zram swap 空间总大小
Swap — Free - zram swap 空间剩余内存大小
Buffer - 物理存储 block 层 buffer 内存
KernStack - 内核栈 KernelStack 内存
Slab - 内核 Slab 内存
Cache — 文件系统 pagecache 内存
Sysfre - 系统 free 内存

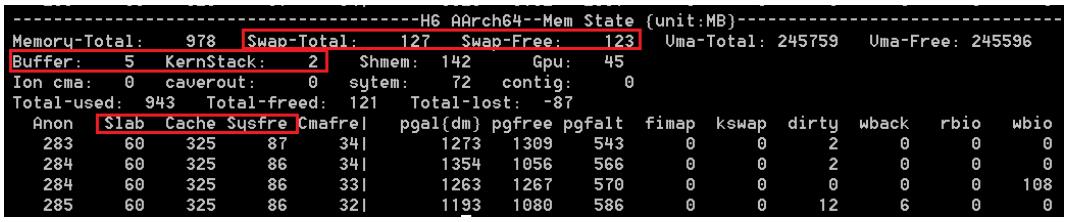


图 4: CpuMonitor 观察内核基础内存分布

- 图形内存 Android 系统，用于多媒体编解码的 buffer 和显示 graphicbuffer 消耗使用的 Ion 内存；Gpu 驱动内部消耗内存。通过 cpu_monitor 工具可以抓取到内核基础内存如下图中所示：

  Ion cma - ION CMA HEAP 内存消耗
  caverout — ION CAVEROUT HEAP 内存消耗
  sytem -ION SYSTEM HEAP 内存消耗

contig -ION CONTIG HEAP 内存消耗
Gpu-GPU 驱动内存消耗

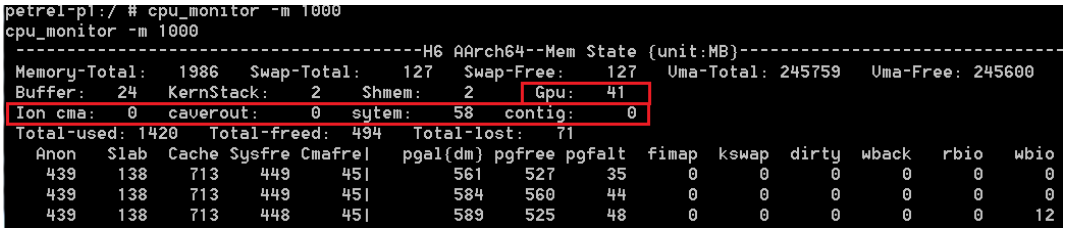


图 5: CpuMonitor 观察内核 ION 内存分布

- 内核私有 page 内存内核基础和驱动模块，有时候会直接或者间接通过 page_alloc() 函数申请 buddy 内存，此部分驱动很难被内核追踪到。通过 cpu_monitor 工具可以抓取到内核基础内存如下图中所示：

Total-lost-内核私有 page 内存消耗总的统计
Total-lost = Memory-Total - Total-used - Total-freed

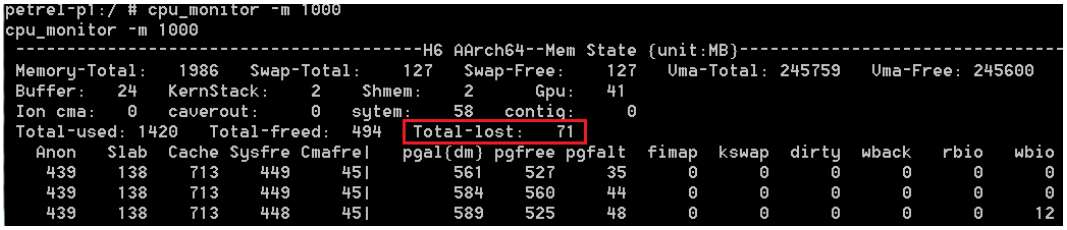


图 6: CpuMonitor 观察内核私有内存泄漏

Android 内存空间内存泄漏主要发生在新 Soc 平台 BSP 开发验证阶段，内存泄漏主要表现在 Slab 内存泄漏、Ion 内存泄漏和内核驱动私有 page 使用泄漏。通过 cpu_monitor 工具确认了系统内核空间存在内存泄漏后，就要通过更有效的工具和方法继续调试分析，以下小节中将重点针对问题的具体类型进行调试分析说明。

3.2.1 内核 slab 内存泄漏

- 内存泄漏场景 任何场景下

- 问题确认方法

```
adb shell mount debugfs -t debugfs /sys/kernel/debug
adb shell cpu_monitor -m 1000
```

工具运行后如下图，需要观察工具定期采样，观察抓取的 Slab 计数的变化，如下图出现的异常增大时，可以确定问题：

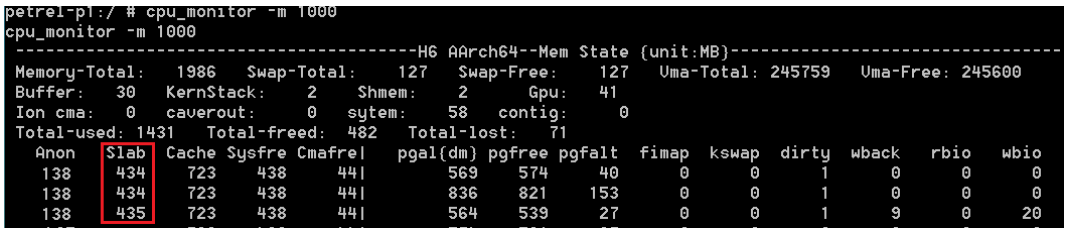


图 7: CpuMonitor 观察 Slab 泄漏

在系统正常启动到主界面后，通过 CpuMonitor 观察到的 slab 数量，一般在 69 ～ 70MB 左右浮动，任何场景的老化经过 12h 或者更久如果发现 slab 数量持续增长，且 stop/start 命令重新 android 系统后，slab 无法降回到开机的水平，基本上可以断定内核空间发生了 slab 内存泄漏。

- 问题调试分析首先将内核的 slab debug 开关打开

```
CONFIG_SLUB_STATS=y
CONFIG_SLUB_DEBUG=y
```

其次编译 slabinfo 工具，linux-3.10/tools/vm/slabinfo.c 文件导入到 android 系统中进行编译，Android.mk 文件如下：

```
LOCAL_PATH := $(call my-dir)
LOCAL_MODULE := slabinfo
LOCAL_SRC_FILES := slabinfo.c
LOCAL_SHARED_LIBRARIES := libcutils
```



```
LOCAL_MODULE_TAGS := optional
LOCAL_ADDITIONAL_DEPENDENCIES := $(LOCAL_PATH)/Android.mk
include $(BUILD_EXECUTABLE)
```

生成在 out/target/product/petrel-p1/system/bin/slabinfo 文件后，导入到机器中运行：

```
adb remount
adb push slabinfo /system/bin
adb shell chmod 777 /system/bin/slabinfo
adb shell slabinfo
```

工具正常运行后如下图：

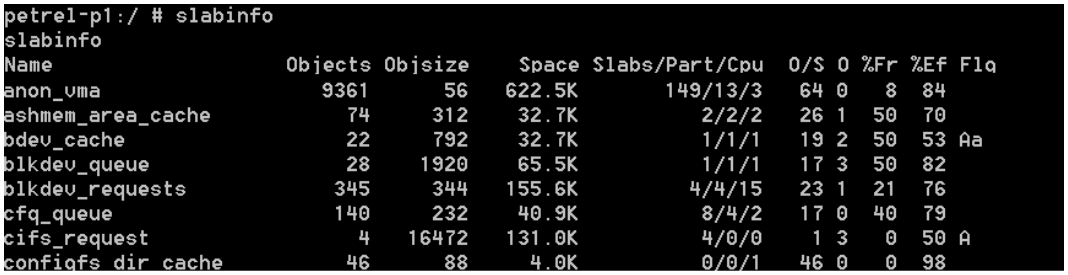


图 8: slabinfo 工具信息

使用 slabinfo 工具每间隔一定时间采样，观察是个 Name 对应的 slab object 在增长。找到 slab 增长点后，进行如下调试：举个例子：假设通过 slabinfo 工具追踪到了 kmalloc-128 区在不断的增长，进入该区对应的 slab sysfs debug 节点分析：

```
adb shell cat /sys/kernel/slab/kmalloc-128/alloc_calls
可以看到打印如下：
root@petrel-p1:/sys/kernel/slab/kmalloc-128 #  cat alloc_calls
  106 kbase_alloc_free_region+0x3c/0x74 [mali_kbase] age=635266/5409565/6776253 pid=1786-10649
   10 kbase_mmap+0x580/0x974 [mali_kbase] age=5799903/6483188/6776252 pid=1786-10649
    1 dhd_deferred_work_init+0x90/0x284 [bcmdhd] age=6775843 pid=2506
   19 mempool_create_node+0x48/0x10c age=6776510/6776717/6776781 pid=1-6
   11 shmem_fill_super+0x38/0x1c4 age=6776204/6776412/6776803 pid=1-1787
    1 strndup_user+0x50/0x74 age=6775937 pid=2506
```



```
     1 pcpu_mem_zalloc+0x60/0x80 age=6776750 pid=1
561481 alloc_vmap_area.isra.29+0xc8/0x348 age=845/6200344/6776810 pid=0-30147
     1 pcpu_get_vm_areas+0x244/0x53c age=6776750 pid=1
   575 __proc_create+0xa4/0xfc age=1839/6741340/6776808 pid=0-3739
     4 kmp_init+0x34/0x1b0 age=6776491/6776491/6776491 pid=1
    17 pinctrl_get+0xf4/0x3ac age=6776264/6776607/6776791 pid=1-1761

     2 sunxi_pinctrl_init+0x39c/0x940 age=6776790/6776790/6776791 pid=1
   124 gpiochip_add_pin_range+0x44/0xf4 age=6776790/6776790/6776791 pid=1
     1 pwmchip_add+0xd8/0x1ac age=6776781 pid=1
     2 sunxi_pwm_probe+0x40/0x6ac age=6776781/6776781/6776781 pid=1
     ... ...
```

通过以上信息可以看到，alloc_vmap_area.isra.29+0xc8/0x348 的申请存在明显异常，极有可能是泄漏的罪魁祸首，因此需要进入更细致的调试，

```
adb shell echo 0 > /proc/sysrq-trigger
adb shell echo 1 > /sys/kernel/slab/kmalloc-128/trace
adb shell cat /proc/kmsg > /data/slab_kmalloc-128-trace.log
adb pull /data/slab_kmalloc-128-trace.log
```

从 slab_kmalloc-128-trace.log 文件中搜索 alloc_vmap_area 关键字，可以搜索到该关键字对应的内核 backtrace 信息，有时候可以直接抓到问题对应的点，例如下图中的信息，后面经过代码 review，确认是此处发生了泄漏。如果有多种 backtrace 类型调用到了该关键字，因此需要耐心分析，一般情况下，最容易出现问题的地方是往往是二次开发过程中内核驱动模块引入。

```
6>[  554.826456] TRACE kmalloc-128 alloc 0xffffffc01f41cc00 inuse=16 fp=0x       (null)
<4>[  554.826463] CPU: 0 PID: 484 Comm: kworker/u9:0 Tainted: G          O 3.10.65 #3
<4>[  554.826503] Workqueue: mali_jd kbase_fence_wait_worker [mali_kbase]
<0>[  554.826507] Call trace:
<4>[  554.826515] [<ffffffc000088768>] dump_backtrace+0x0/0x11c
<4>[  554.826521] [<ffffffc0000888a4>] show_stack+0x20/0x30
<4>[  554.826527] [<ffffffc0007df664>] dump_stack+0x1c/0x28
<4>[  554.826533] [<ffffffc0007dd520>] trace.part.55+0x74/0x88
<4>[  554.826540] [<ffffffc00018fbfc>] alloc_debug_processing+0x140/0x168
<4>[  554.826546] [<ffffffc000190d2c>] __slab_alloc.isra.59.constprop.66+0x33c/0x3b4
<4>[  554.826552] [<ffffffc000191210>] kmem_cache_alloc_trace+0x7c/0x1d4
```

```
<4>[  554.826559] [<ffffffc0001822f8>] alloc_vmap_area.isra.29+0xc4/0x348
<4>[  554.826565] [<ffffffc000182f90>] __get_vm_area_node.isra.30+0x104/0x1a4
<4>[  554.826572] [<ffffffc000183ad0>] get_vm_area_caller+0x44/0x58
<4>[  554.826578] [<ffffffc00009700c>] __ioremap_caller+0x88/0xd4
<4>[  554.826583] [<ffffffc000097090>] __ioremap+0x38/0x4c
<4>[  554.826617] [<ffffffbffc09c5b4>] set_reg_bit.constprop.6+0x2c/0x5c [mali_kbase]
<4>[  554.826647] [<ffffffbffc09c62c>] sunxi_protected_mode_reset +0x1c/
0x2c [mali_kbase]
<4>[  554.826678] [<ffffffbffc0a5510>] kbase_pm_init_hw+0xd0/0x3e0 [mali_kbase]
<4>[  554.826708] [<ffffffbffc0a58ac>] kbase_pm_clock_on+0x8c/0x11c [mali_kbase]
<4>[  554.826739] [<ffffffbffc0a3e24>] kbase_pm_do_poweron+0x24/0x3c [mali_kbase]
<4>[  554.826771] [<ffffffbffc0a6788>] kbase_pm_update_active
+0xc4/0x160 [mali_kbase]
<4>[  554.826802] [<ffffffbffc0a4200>] kbase_hwaccess_pm_gpu_active +0x1c/
0x2c [mali_kbase]
<4>[  554.826826] [<ffffffbffc0896bc>] kbase_pm_context_active_handle_suspend
+0x88/0xb0 [mali_kbase]
<4>[  554.826850] [<ffffffbffc08776c>] kbase_js_sched+0x144/0x71c [mali_kbase]
<4>[  554.826875] [<ffffffbffc08d1a0>] kbase_fence_wait_worker
+0x68/0x84 [mali_kbase]
<4>[  554.826884] [<ffffffc0000b8140>] process_one_work+0x270/0x3f0
<4>[  554.826890] [<ffffffc0000b9294>] worker_thread+0x210/0x330
<4>[  554.826897] [<ffffffc0000bf158>] kthread+0xb4/0xc0
```

3.2.2 内核私有 page 使用泄漏

- 内存泄漏场景 任何场景下
- 问题确认方法

```
adb shell mount debugfs -t debugfs /sys/kernel/debug
adb shell cpu_monitor -m 1000
```

工具运行后如下图，需要观察工具抓取的 Total-lost 计数的变化，如果持续增加，并且 stop/start Android 系统后仍无法恢复到系统开机阶段的正常水准，可以确定内核存在 page alloc 内存泄漏！

图 9: CpuMonitor 观察内核私有内存泄漏

- 问题调试分析在内核打上 msm-3.10 page owner patch，专门用于分析内核的私有页面申请

  0001-debugging-keep-track-of-page-owners.patch
  0002-page-owners-correct-page-order-when-to-free-page.patch
  重新编译内核时会提示选择 CONFIG_PAGE_OWNER=Y/N？选择 Y

编译 page owner 配套使用的工具 linux-3.10/Documentation/page_owner.c 文件导入到 android 系统中进行编译，Android.mk 文件如下：

```
LOCAL_PATH := $(call my-dir)
LOCAL_MODULE := sort
LOCAL_SRC_FILES :=page_owner.c
LOCAL_SHARED_LIBRARIES := libcutils
LOCAL_MODULE_TAGS := optional
LOCAL_ADDITIONAL_DEPENDENCIES := $(LOCAL_PATH)/Android.mk
include $(BUILD_EXECUTABLE)
```

生成在 out/target/product/petrel-p1/system/bin/sort 文件后，导入到机器中运行：

```
adb remount
adb push sort /system/bin
adb shell chmod 777 /system/bin/sort
```


重新进行问题老化实验，使用 cpumonitor 工具确认了 Total － LOST 内存已经累积到一定量后，进行如下操作：

```
adb shell stop
adb shell start
adb shell mount debugfs -t debugfs /sys/kernel/debug
adb shell cat /sys/kernel/debug/page_owner > /data/page_owner_full.txt
adb shell grep -v ^PFN page_owner_full.txt > /data/page_owner.txt
adb shell sort /data/page_owner.txt /data/sorted_page_owner.txt
adb pull /data/sorted_page_owner.txt
```

打开 sorted_page_owner.txt 文件，观察该文件信息，找到最大的内存申请次数点 times 信息，根据 order 大小计算该内存占据的量是否和 Total － LOST 接近或者说占绝大比例量。例如下面的 log 中是在某次问题案例中抓取的内存泄漏点，泄漏大小为 119706×4KB ＝ 460MB（order ＝ 0,4KB 页面，order ＝ 1,8KB 页面，order 大小为 2 的幂次方），通过占据内存的 backtrace 找到问题点。

```
119706 times
Page allocated via order 0, mask 0x2084d0
[<ffffffc00015b700>] __get_free_pages+0x30/0x80
[<ffffffc00017596c>] __pte_alloc_kernel+0x28/0xa8
[<ffffffc000183514>] vmap_page_range_noflush+0x218/0x284
[<ffffffc0001845d8>] vm_map_ram+0x80/0xb0
[<ffffffc0005b6490>] ion_heap_clear_pages+0x34/0x74
[<ffffffc0005b6580>] ion_heap_sglist_zero+0xb0/0xe8
[<ffffffc0005b6974>] ion_heap_buffer_zero+0x44/0x54
[<ffffffc0005b7994>] ion_system_heap_free+0x50/0xdc
```

3.2.2 内核 Ion 内存泄漏

- 内存泄漏场景

第三方播放器本地视频播放；
第三方直播应用、在线视频播放应用

- 问题确认方法

```
adb shell mount debugfs -t debugfs /sys/kernel/debug
adb shell cpu_monitor -m 1000
```

工具运行后如下图，需要观察工具抓取的 ion 一栏各项计数变化，如 H6 iommu 内存方案使用 ion system heap 非连续物理内存，只需要观察 sytem 计数变化，如下图发现察 sytem 计数出现明显异常时，需要继续确认内存泄漏：

```
----------------------------------------H6 AArch64--Mem State {unit:MB}-------------------------------
Memory-Total:   1986   Swap-Total:    127   Swap-Free:    126   Vma-Total: 245759   Vma-Free: 245593
Buffer:   15   KernStack:    2    Shmem:    6      Gpu:   66
Ion cma:   0   caverout:     0   sytem:   746   contig:     0
Total-used: 1793   Total-freed:   92   Total-lost:  100
  Anon   Slab  Cache Sysfre Cmafre|   pgal{dm} pgfree pgfalt  fimap  kswap  dirty  wback   rbio   wbio
   428     92    433     89      2|      2124   1877     20      0     13      0     15      0     20
   428     92    434     88      2|      2232   1978     73      0     14      1      0     52     56
   428     92    432     90      2|      2348   2768     22      0     32     12      0      0      0
```

图 10: CpuMonitor 观察 ION 内存泄漏

- 问题调试分析

```
mount debugfs -t debugfs /sys/kernel/debug
cat /sys/kernel/debug/ion/heaps/system
```

如下图中数据，client 列对应的进程名表示内存申请创建者信息，pid 对应的数字是进程的 pid，size 表示内存创建大小，单位 byte。绿色区域表示 ion system heap 正常使用的内存，红色区域 orphaned allocations 表示 ion system heap 泄漏内存，橙色区域中 total orphaned 表示当前系统泄漏的 ion system hea 内存总数，total 表示当前系统已经申请的 ion system hea 内存总数，即正在使用的 + 已经泄漏的总和。

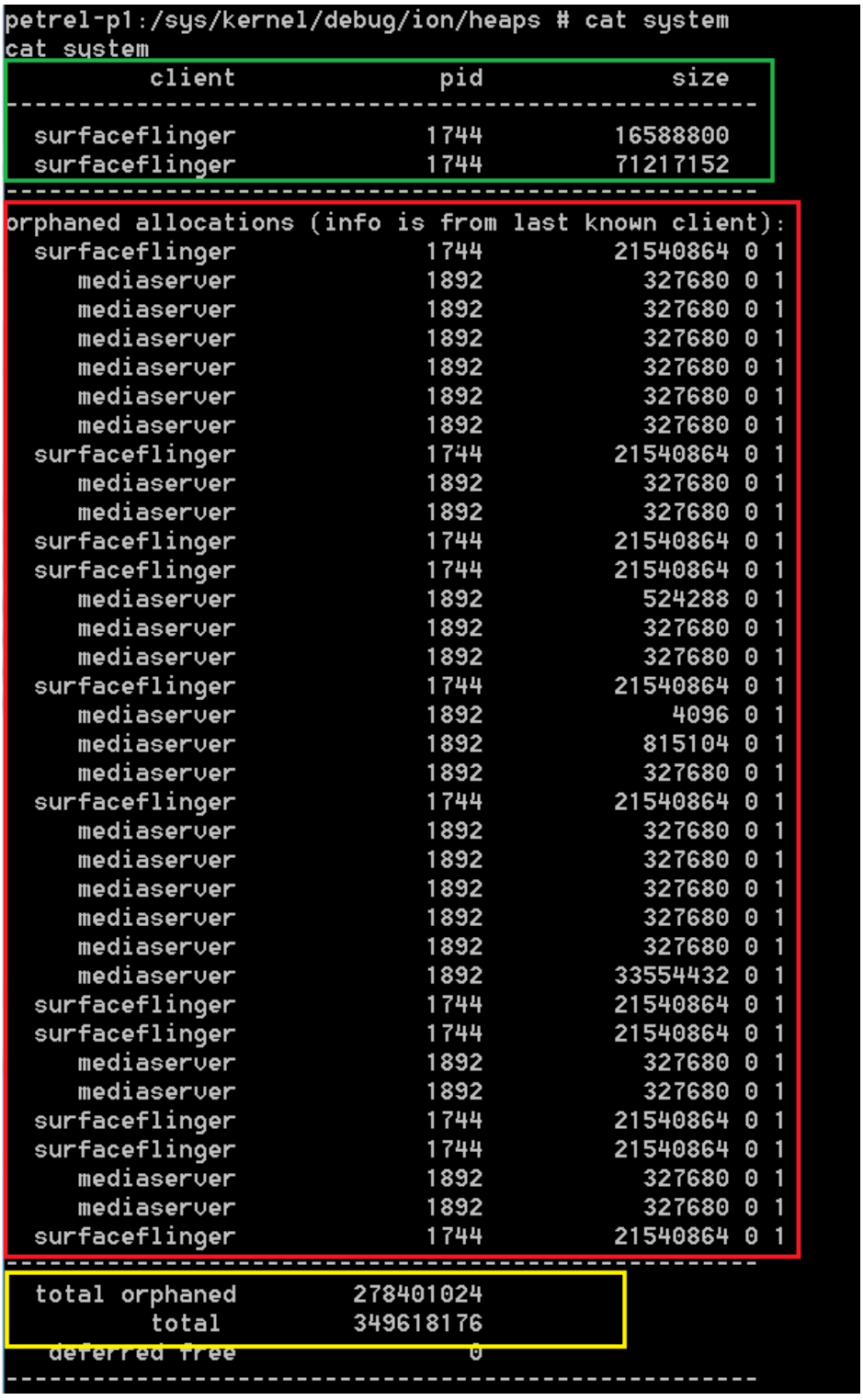


图 11: ionsystemheap 内存泄露信息

多媒体场景下，mediaserver 进程空间运行着解码器线程和往播放器的 surfaceview 图层不断填充解码数据的 Render 线程，前者在多媒体视频文件解码的过程中需要从 ion 中申请创建 buffer，对应着上图红色区域中 mediaserver；送显 buffer 是从通过申请 ANativeWindow 实例的方式，从 surfaceflinger 进程空间的 gralloc 分配的 graphic buffer，对应这上图红色区域中的 surfaceflinger 进程。多媒体场景下 ion system heap 内存泄漏主要有两种方法：

mediaserver 进程的解码器线程和送显线程没有 unmap 解除 dma buffer 映射；
mediaserver 进程的解码器线程和送显线程没有 close 释放 dma buffer fd 句柄；

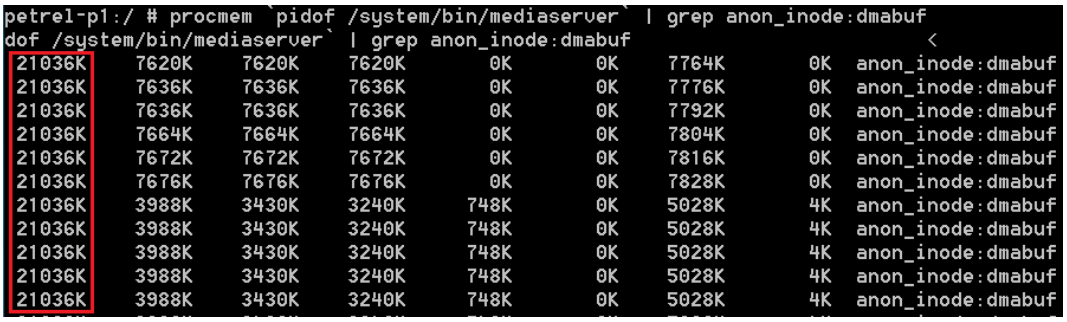
通过以下命令，可以查看泄漏的 dma buffer fd，在 Android7.0 系统上，会发现找不到泄漏的句柄，此时极有可能是 mediaserver 进程异常退出，没有释放 dma buffer fd 句柄导致，可以通过检查红色区域 orphaned allocations 中 client mediaserver pid 信息，是否和当前运行的 mediaserver 进程 pid 一致来确认，如果不一致，说明 mediaserver 进程出现过异常退出，因此需要进一步检查下/data/tombstone/目录下的文件信息，分析 mediaserver 异常退出原因。

```
lsof -p `pidof /system/bin/mediaserver` | grep anon_inode:dmabuf
```

图 12: ion dmabuffer 句柄泄露信息

通过以下命令，可以查看泄漏的 dma buffer 映射

```
procmem `pidof /system/bin/mediaserver` | grep anon_inode:dmabuf
```

```
petrel-p1:/ # procmem `pidof /system/bin/mediaserver` | grep anon_inode:dmabuf
dof /system/bin/mediaserver` | grep anon_inode:dmabuf                        <
 21036K    7620K    7620K    7620K       0K       0K    7764K       0K  anon_inode:dmabuf
 21036K    7636K    7636K    7636K       0K       0K    7776K       0K  anon_inode:dmabuf
 21036K    7636K    7636K    7636K       0K       0K    7792K       0K  anon_inode:dmabuf
 21036K    7664K    7664K    7664K       0K       0K    7804K       0K  anon_inode:dmabuf
 21036K    7672K    7672K    7672K       0K       0K    7816K       0K  anon_inode:dmabuf
 21036K    7676K    7676K    7676K       0K       0K    7828K       0K  anon_inode:dmabuf
 21036K    3988K    3430K    3240K     748K       0K    5028K       4K  anon_inode:dmabuf
 21036K    3988K    3430K    3240K     748K       0K    5028K       4K  anon_inode:dmabuf
 21036K    3988K    3430K    3240K     748K       0K    5028K       4K  anon_inode:dmabuf
 21036K    3988K    3430K    3240K     748K       0K    5028K       4K  anon_inode:dmabuf
 21036K    3988K    3430K    3240K     748K       0K    5028K       4K  anon_inode:dmabuf
```

图 13: ion dmabuffer 映射泄露信息

上图中红色部分的 dmabuf map 空间大小，需要确认是否和 orphaned allocations 信息中的 buffer 大小一致，如果一致，基本可以确认 dmabuf 泄漏位于 mediaserver 进程空间，接下来可以通过在 mediaserver 代码中所有申请 ion buffer、释放 ion buffer、解除 dmabuf mmap 和 close dmabuf fd 的代码点，添加 debug 信息，即可找到泄漏的点。

如果以上方法还是无法分析追踪到 ion system heap 内存泄漏点，建议使用终极武器全面分析，找到内存泄漏必现的规律和场景，执行以下命令对 ion 进行 trace 数据分析，从抓取的信息中进一步找到线索。

```
adb remount
adb shell "mount debugfs -t debugfs /sys/kernel/debug"
adb shell "echo 1 > /sys/kernel/debug/tracing/events/tracing_on"
adb shell "echo 1 > /sys/kernel/debug/tracing/events/ion/enable"
adb shell "cat /sys/kernel/debug/tracing/trace_pipe"
```

```
surfaceflinger-1743  [001] ...1  6833.921307: ion_map: handle id:3 dma_buf fd:69
 Binder:1743_4-2685  [002] ...1  6833.925469: ion_free_handle:  handle id 26 free ,buffer:ffffffc075c07f00 handle_count 0
 Binder:1743_4-2685  [002] ...1  6833.925477: ion_free_buffer: buffer:ffffffc075c07f00
 Binder:1743_4-2685  [002] ...1  6833.925786: ion_free_handle:  handle id 25 free ,buffer:ffffffc05a037700 handle_count 0
 Binder:1743_4-2685  [002] ...1  6833.925790: ion_free_buffer: buffer:ffffffc05a037700
 Binder:1743_4-2685  [002] ...1  6833.925935: ion_alloc: len 8294400 align 0 heap:1 flags 0 handle id 25 buffer ffffffc075c07f00
 Binder:1743_4-2685  [002] ...1  6833.925984: ion_map: handle id:25 dma_buf fd:87
 Binder:1743_4-2685  [002] ...1  6833.929377: ion_alloc: len 4096 align 0 heap:16 flags 0 handle id 26 buffer ffffffc05d436100
 Binder:1743_4-2685  [002] ...1  6833.929467: ion_map: handle id:26 dma_buf fd:90
surfaceflinger-1743  [001] ...1  6833.937594: ion_import: new handle id:2 dma_buf fd:36
surfaceflinger-1743  [001] ...1  6833.937628: ion_map: handle id:2 dma_buf fd:112
surfaceflinger-1743  [001] ...1  6833.937644: ion_import: new handle id:4 dma_buf fd:62
surfaceflinger-1743  [001] ...1  6833.937652: ion_map: handle id:4 dma_buf fd:42
surfaceflinger-1743  [001] ...1  6833.937657: ion_import: new handle id:3 dma_buf fd:71
surfaceflinger-1743  [001] ...1  6833.937664: ion_map: handle id:3 dma_buf fd:88
surfaceflinger-1743  [001] ...1  6833.954527: ion_import: new handle id:2 dma_buf fd:36
surfaceflinger-1743  [001] ...1  6833.954567: ion_map: handle id:2 dma_buf fd:113
surfaceflinger-1743  [001] ...1  6833.954584: ion_import: new handle id:4 dma_buf fd:62
surfaceflinger-1743  [001] ...1  6833.954592: ion_map: handle id:4 dma_buf fd:32
surfaceflinger-1743  [001] ...1  6833.954596: ion_import: new handle id:3 dma_buf fd:71
surfaceflinger-1743  [001] ...1  6833.954602: ion_map: handle id:3 dma_buf fd:86
 Binder:1743_4-2685  [002] ...1  6833.962217: ion_free_handle:  handle id 16 free ,buffer:ffffffc056b21c00 handle_count 0
 Binder:1743_4-2685  [002] ...1  6833.962228: ion_free_buffer: buffer:ffffffc056b21c00
 Binder:1743_4-2685  [002] ...1  6833.962526: ion_free_handle:  handle id 15 free ,buffer:ffffffc05a037600 handle_count 0
```

图 14: ion trace 跟踪信息

当然，如果只是简单的追踪进程空间的 ion buffer 的申请和释放，可以通过以下命令对 ion trace 信息进行瘦身：

```
adb remount
adb shell "mount debugfs -t debugfs /sys/kernel/debug"
adb shell "echo > /sys/kernel/debug/tracing/set_event"
adb shell "echo ion_alloc>> /sys/kernel/debug/tracing/set_event"
adb shell "echo ion_free_buffer >> /sys/kernel/debug/tracing/set_event"
adb shell "echo ion_free_handle >> /sys/kernel/debug/tracing/set_event"
adb shell "cat /sys/kernel/debug/tracing/trace_pipe"
```

图 15: ion trace 跟踪信息

4. Declaration

This document is the original work and copyrighted property of Allwinner Technology (“Allwinner”). Reproduction in whole or in part must obtain the written approval of Allwinner and give clear acknowledgement to the copyright owner. The information furnished by Allwinner is believed to be accurate and reliable. Allwinner reserves the right to make changes in circuit design and/or specifications at any time without notice. Allwinner does not assume any responsibility and liability for its use. Nor for any infringements of patents or other rights of the third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of Allwinner. This datasheet neither states nor implies warranty of any kind, including fitness for any particular application.tates nor implies warranty of any kind, including fitness for any particular application.

