Hang Detect 问题梳理

Haiquan

2019-11-27

目录

- 背景介绍
- 源码路径
- 工作原理
- Debug 方法
- 案例分享

背景介绍：

1、机器死机或变砖，给消费者带来非常恶劣的用户体验

2、厂家难以分析

3、传统的设计存在缺陷

那么，之前是如何设计死机保护呢？

在Google Android 系统中, 死机保护是通过watchdog 机制来达成, 即将死机转换成重启.

- **HW Watchdog**

  用于监测CPU 执行是否异常, 启用Kernel RT thread tick HW watchdog 来达成, 如果异常, 则重启整个系统.

- **System Server Watchdog**

  用于监测Android System Server 关键线程和资源使用是否正常, 如果异常则重启android 上层.

这样设计死机保护，存在什么样的缺陷呢？

- HW Watchdog 和 System Server Watchdog 分离执行
- System Server Watchdog 依赖于底层本身稳定性
- System Server 可能重启fail 掉, 导致一直卡死

这样的缺陷，可能会导致什么问题呢？

导致System Server 卡死或者重启失败，其场景如下：

上层：

- Android 虚拟机(ART/Dalvik) 异常, 导致Java 层代码无法执行, System Server 卡死
- Surfaceflinger 卡在, system server 重启后, 将无法重启成功
- System Server Watchdog 抓取资讯时，如果自己卡住，那么将无法重启, 持续卡死

低层：

- File System 异常, 导致System Server Watchdog 卡在, 无法重启
- Memory leaks, 导致System Server 卡在Kernel 中等待memory
- Kernel Driver 异常, 导致system server 无法重启成功

源码路径：

Kernel：

/kernel-3.18/drivers/misc/mediatek/aee/aed/monitor_hang.c
/kernel-3.18/drivers/misc/mediatek/aee/aed/aed.h

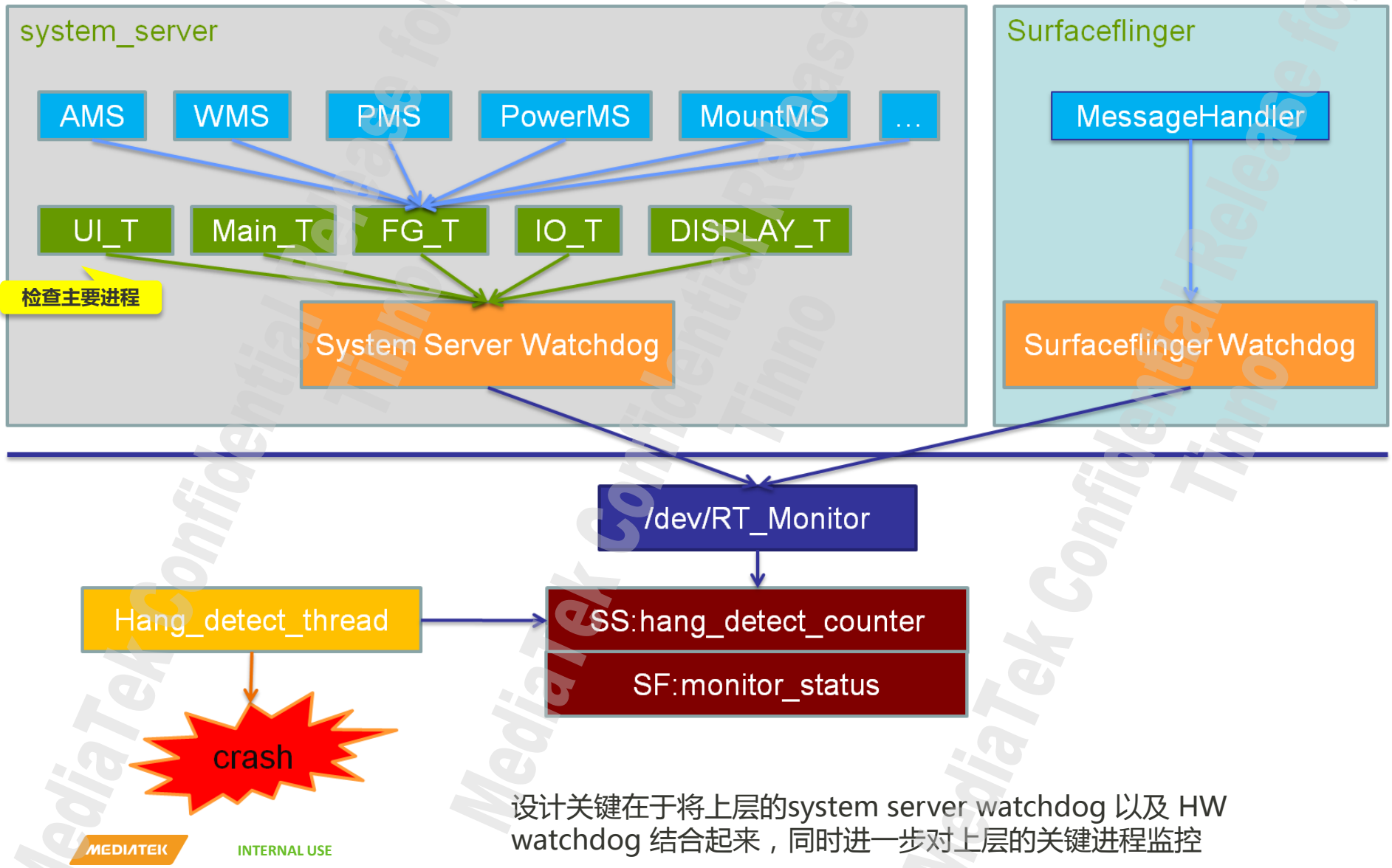
Native：

/vendor/mediatek/proprietary/external/aee/binary/inc/aee.h

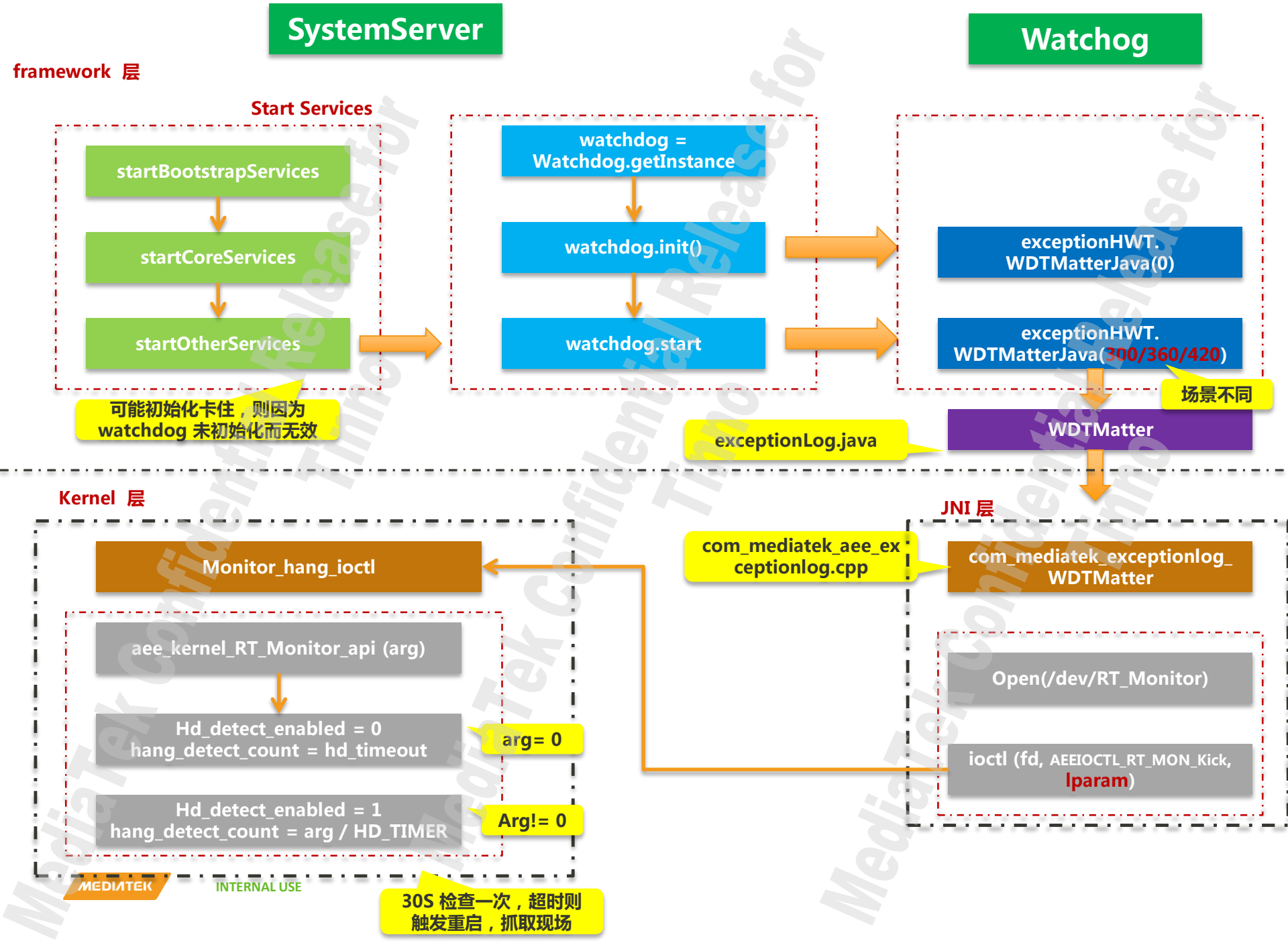
Framwork:

/frameworks/base/services/java/com/android/server/SystemServer.java
/frameworks/base/services/core/java/com/android/server/Watchdog.java
/frameworks/base/core/java/com/mediatek/aee/jni/com_mediatek_aee_exceptionlog.cpp
/frameworks/native/services/surfaceflinger/mediatek/SurfaceFlingerWatchDog.cpp

设计原理



设计关键在于将上层的system server watchdog 以及 HW watchdog 结合起来，同时进一步对上层的关键进程监控


SystemServer
Watchog
framework 层
Start Services
startBootstrapServices
startCoreServices
startOtherServices
可能初始化卡住，则因为
watchdog 未初始化而无效
watchdog =
Watchdog.getInstance
watchdog.init()
watchdog.start
exceptionHWT.
WDTMatterJava(0)
exceptionHWT.
WDTMatterJava(300/360/420)
场景不同
WDTMatter
exceptionLog.java
Kernel 层
JNI 层
com_mediatek_aee_ex
ceptionlog.cpp
com_mediatek_exceptionlog_
WDTMatter
Monitor_hang_ioctl
aee_kernel_RT_Monitor_api (arg)
Hd_detect_enabled = 0
hang_detect_count = hd_timeout
arg= 0
Hd_detect_enabled = 1
hang_detect_count = arg / HD_TIMER
Arg!= 0
Open(/dev/RT_Monitor)
ioctl (fd, AEEIOCTL_RT_MON_Kick,
lparam)
30S 检查一次，超时则
触发重启，抓取现场
MEDIATEK
INTERNAL USE

```
while (1) {
        if ((1 == hd_detect_enabled) && (FindTaskByName("system_server") != -1)) {
                LOGE("[Hang_Detect] hang_detect thread counts down %d:%d. \n", hang_detect_counter, hd_timeout);

                if (hang_detect_counter <= 0)
                        ShowStatus();

                if (hang_detect_counter == 0) {
                        LOGE("[Hang_Detect] we should triger    HWT     ...\n");
                        if (aee_mode != AEE_MODE_CUSTOMER_USER) {
                                aee_kernel_warning_api
                                        (__FILE__, __LINE__,
                                        DB_OPT_NE_JBT_TRACES | DB_OPT_DISPLAY_HANG_DUMP,
                                        "\nCRDISPATCH_KEY:SS Hang\n",
                                        "we triger HWT ");
                                msleep(30 * 1000);
                        } else {
                                /* only Customer user load,Only triger KE  */
                                BUG();
                        }
                }

                hang_detect_counter--;
```

每30S 检查一次

如果timeout 则打印各线程的backtrace

非User 版本，触发HWT

User 版本触发Bug

SSW 在执行的过程中，会周期性的tick hang detect,

tick 分成四种，后续版本将不断优化优化.

- 正常情况下，tick 300s, 对应count=10.
- 在dump backtrace 时，tick 600s, 对应count=20.
- 在SWT 发生的情况下，tick 720s, 对应count=24.
- 当surfaceflinger/system server 发生NE
  因为coredump 抓取比较耗时, MTK aee 会主动tick 660s, 对应count=22

SW	场景	TICK	Hang_detect_count
N & N1 & O以后	正常情况	300S	300/30 = 10
N & N1 & O以后	Dump backtrace	360S	12
N & N1 & O以后	SWT	420S	14
N1 & O以后	抓取AEE DB	330S	11
O以后	SS/SF 发生NE	600S	20
O以后	SS/SF 发生NE，开始抓取coredump	1200S	40
O以后	SS/SF 发生NE，抓取完coredump后	570S	19

Hang_detect Debug 思路

Hang Detect 问题发生时，意味着watchdog 没有正确的执行。

按照如下方法分析：

- 确认SSW tick Hang Detect的状态
- 确定SSW Thread 状态
- 关键process 状态追查
- Hang Detect KE 调整
- 保存现场快速分析

1、确认SSW Tick Hang Detect 的状态

可以从kernel log 中明确的看到:

- watchdog 看起来正常：

```
[  198.215932] (1)[1322:watchdog]AEEIOCTL_RT_MON_Kick ( 300)
[  198.215945] (1)[1322:watchdog][Hang_Detect] hang_detect enabled 10
```

- watchdog 在dump backtrace:

```
[  258.218145] (0)[1322:watchdog]AEEIOCTL_RT_MON_Kick ( 600)
[  258.218171] (0)[1322:watchdog][Hang_Detect] hang_detect enabled 20
```

- watchdog 在做SWT:

```
[  299.046542] (0)[1322:watchdog]AEEIOCTL_RT_MON_Kick ( 720)
[  299.046572] (0)[1322:watchdog][Hang_Detect] hang_detect enabled 24
```

当然也可以从hang detect thread 的log 中看到这个：

```
[  210.475572] (0)[90:hang_detect][Hang_Detect] init found pid:1.
[  210.475735] (0)[90:hang_detect][Hang_Detect] mmcqd/0 found pid:158.
[  210.475815] (0)[90:hang_detect][Hang_Detect] surfaceflinger found pid:265.
[  210.475887] (0)[90:hang_detect][Hang_Detect] system_server found pid:734.
[  210.475919] (0)[90:hang_detect][Hang_Detect] ndroid.systemui found pid:1071.
[  210.476003] (0)[90:hang_detect][Hang_Detect] debuggerd found pid:4313.
[  210.476027] (0)[90:hang_detect][Hang_Detect] debuggerd64 found pid:4314.
[  210.476056] (0)[90:hang_detect][Hang_Detect] hang_detect thread counts down 10:10.
```

2、确认SSW Thread 的状态

- **Watchdog 正常时，对应的backtrace.**

```
KERNEL SPACE BACKTRACE, sysTid=1299

   [] __switch_to+0x74/0x8c from []
   [] __schedule+0x314/0x794 from []
   [] schedule+0x24/0x68 from []
   [] futex_wait_queue_me+0xcc/0x158 from []
   [] futex_wait+0x120/0x20c from []
   [] do_futex+0x184/0xa48 from []
   [] SyS_futex+0x88/0x19c from []
   [] cpu_switch_to+0x48/0x4c from []

"watchdog" sysTid=1299

  #00 pc 0000000000019478  /system/lib64/libc.so (syscall+28)
  #01 pc 00000000000d0c74  /system/lib64/libart.so (art::ConditionVariable::TimedWait(art::Thread*, long, int)+168)
  #02 pc 00000000002a3e98  /system/lib64/libart.so (art::Monitor::Wait(art::Thread*, long, int, bool, art::ThreadState)+860)
  #03 pc 00000000002a4e04  /system/lib64/libart.so (art::Monitor::Wait(art::Thread*, art::mirror::Object*, long, int, bool, art::ThreadState)+244)
  #04 pc 0000000000000974  /data/dalvik-cache/arm64/system@framework@boot.oat
```

- **不是这个backtrace，则说明watchdog 已经卡住**

- **如果你没有找到watchdog 线程怎么办呢？**

此时可能是两种情况：

1、system server SWT 重启后，system server 初始化失败
watchdog 还没有启动就卡死了。

2、system server 发生SWT 后，退出
因为卡在了kernel 导致system server 某个线程无法退出，分析此线程。

3、关键Process 状态追查

根据以往分析的经验，发现很多时候都是一些关键的process 卡住，导致watchdog 的行为无法达成. 所以在hang detect 中打印了关键的Process 的资讯，以便快速追查问题.

```
[  300.505542] (0)[90:hang_detect][Hang_Detect] init found pid:1.
[  300.505616] (0)[90:hang_detect][Hang_Detect] mmcqd/0 found pid:158.
[  300.505649] (0)[90:hang_detect][Hang_Detect] surfaceflinger found pid:265.
[  300.505678] (0)[90:hang_detect][Hang_Detect] system_server found pid:734.
[  300.505688] (0)[90:hang_detect][Hang_Detect] ndroid.systemui found pid:1071.
[  300.505719] (0)[90:hang_detect][Hang_Detect] debuggerd found pid:4313.
[  300.505726] (0)[90:hang_detect][Hang_Detect] debuggerd64 found pid:4314.
[  300.505735] (0)[90:hang_detect][Hang_Detect] hang_detect thread counts down 0:24.
```

线程卡住	说明
Init	任何system property 的设置或则service的重启wifi 的设置都会卡住
mmcqd	emmc 可能卡住
Surfaceflinger	Display dirver 和 GPU 可能卡住
systemui	Keyguard 可能卡住
debuggerd	Watchdog 抓取backtrace 时就会卡住

4、Hang Detect KE 调整

- **SYS_HANG_DETECT_RAW**
  开辟专门的hang detect buffer 存储相关进程信息；不担心被刷掉

- **DATA_ANR_TRACES**
  查看上层的backtrace, 存放/data/anr 的资讯
  再确认到system server 的SWT 流程已经抓上层trace 时, 查看

5、保存线程分析

- 当前面的资讯都难以最终定位root cause

- 将重启再变成死机，再用分析死机现场

案例分享

Case1： ALPS04678125 [[CF Blocking][Moto][Lima] 手机卡死，按电源键无法点亮屏幕，插上充电器没有反应]

平　　台：MT6771
软件版本：alps-mp-p0.mp1.tc2sp-V1.17

现象：

按Pwrkey 无反应，插入USB也没有反应，adb 也无法使用；只能长按Pwrkey 触发重启；抓取不到有效的log信息。

SYS_HANG_DETECT_RAW文件中，为什么 有些进程在抓取不到？

watchdog D 171.126230 1667 75 11 0x414040 ....
....
<ffffff8008e48c3c> schedule+0x6c/0x88
<ffffff80081539f8> __refrigerator+0x74/0x1ac
<ffffff800817175c> futex_wait_queue_me+0x15c/0x164
<ffffff800816f318> futex_wait+0x154/0x33c
<ffffff800816dccc> do_futex+0xf8/0x15f0
<ffffff8008170e10> SyS_futex+0x150/0x1a8
<ffffff8008085c18> __sys_trace_return+0x0/0x4
<ffffffffffffffff> 0xffffffffffffffff

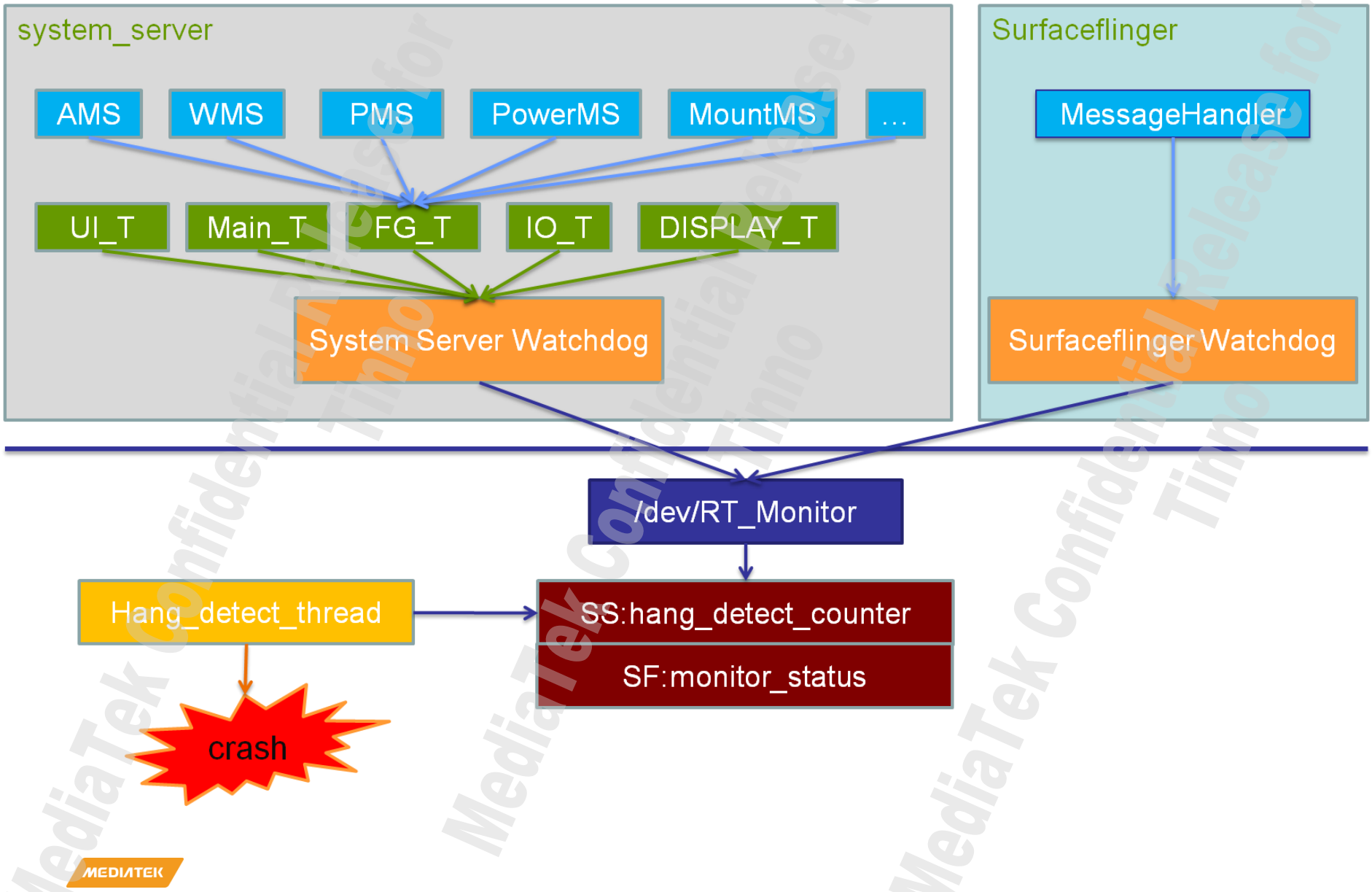
PhotonicModulat D 296740.687611 3822 37750 ...
...
<ffffff8008e4928c> __schedule+0x634/0x8c4
<ffffff8008e48c3c> schedule+0x6c/0x88
<ffffff800813d4d4> synchronize_irq+0x9c/0xec
<ffffff8008145a20> suspend_device_irqs+0x94/0x108
<ffffff80084ff220> dpm_suspend_noirq+0x164/0x690
<ffffff8008134ea0> suspend_devices_and_enter+0x1dc/0xbbc
<ffffff8008136324> **pm_suspend+0x934/0x9dc**
<ffffff8008133c2c> state_store+0x5c/0x98
<ffffff800845b054> kobj_attr_store+0x14/0x24
<ffffff80082db8e8> sysfs_kf_write+0x78/0xac
<ffffff80082da2ac> kernfs_fop_write+0x17c/0x1d4
<ffffff800824631c> vfs_write+0xd4/0x244
<ffffff8008246594> SyS_write+0x54/0xb4
<ffffff8008085c18> __sys_trace_return+0x0/0x4
<ffffffffffffffff> 0xffffffffffffffff

irq/75-primary_ D 100.817142 243 337 0 ....
.....
<ffffff8008e4c424> schedule_timeout+0x40/0x470
<ffffff8008e4b7c0> __down_common+0x94/0xfc
<ffffff8008e4b698> __down+0x14/0x1c
<ffffff8008128a48> down+0x48/0x4c
<ffffff8008a927fc> rt9471_irq_handler+0x70/0x1fc
<ffffff800813fb5c> irq_thread_fn+0x2c/0x50
<ffffff800813f9f4> irq_thread+0x148/0x22c
<ffffff80080d2108> kthread+0xf0/0x100
<ffffff8008085b70> ret_from_fork+0x10/0x20
<ffffffffffffffff> 0xffffffffffffffff

watchdog D 2821.441193 5922 1766 ....
<c0e03cb8> __schedule+0x368/0x910
<c0e042b4> schedule+0x54/0xc4
<c01bd7dc> __refrigerator+0x90/0x188
<c01d8168> futex_wait_queue_me+0x1b4/0x1bc
<c01d8ac8> futex_wait+0x13c/0x29c
<c01da850> do_futex+0x138/0xc88
<c01db4c8> SyS_futex+0x128/0x1ac
<ffffffff> 0xffffffff

irq/89-primary_ D 1482.711631 244 11409 ....
<c0e03cb8> __schedule+0x368/0x910
<c0e042b4> schedule+0x54/0xc4
<c0e07660> schedule_timeout+0x178/0x298
<c0e06410> __down+0x78/0xc4
<c01a1c38> down+0x4c/0x60
<c0a6f6e0> rt9471_i2c_block_read.constprop.3+0x40/0x78
<c0a6f76c> rt9471_irq_handler+0x54/0x154
<c01af4ac> irq_thread_fn+0x24/0x5c
<c01af6d0> irq_thread+0x168/0x288
<c0149aa8> kthread+0x114/0x12c
<ffffffff> 0xffffffff

为什么 pm_suspend backtrace 本次 Hang_detect 未抓取到？



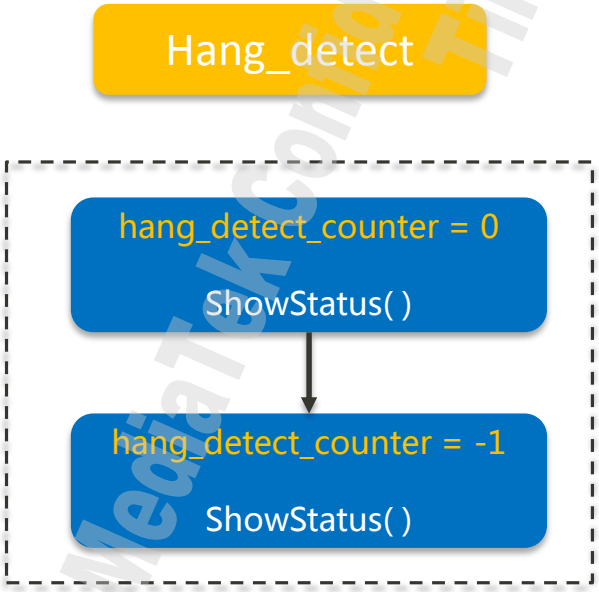
发生Hang_detect 有两种可能：

1、卡在上层

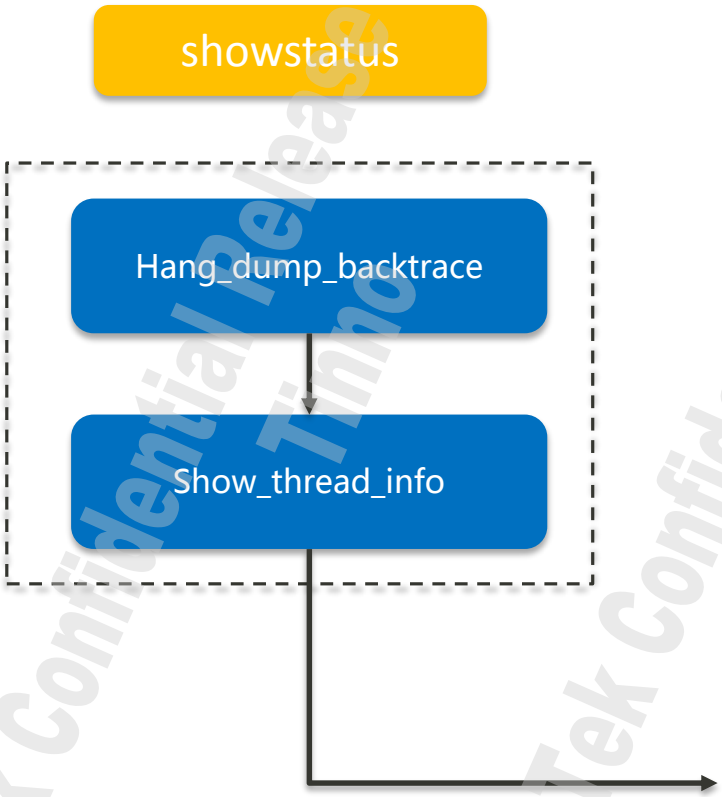
2、卡在Kernel 层

发生异常时，则需要打印关键进程的backtrace进行分析；SYS_HANG_DETECT_RAW 应运而生。

1、何时dump 呢？



2、dump 哪些关键的信息呢？




```
for_each_thread(p, t) {
        if (try_get_task_stack(t)) {
                get_task_struct(t);
                show_thread_info(t, false);
                put_task_stack(t);
                put_task_struct(t);
        }
}
```

```
if (dump_bt || ((p->state == TASK_RUNNING ||
                 p->state & TASK_UNINTERRUPTIBLE) &&
                !strstr(p->comm, "wdtk")))
/* Catch kernel-space backtrace */
        get_kernel_bt(p);
```

3、dump buffer 有多大？

有2M的内存空间存放信息，单条信息可存放256 Byte

```
#ifdef HANG_LOW_MEM
#define MAX_HANG_INFO_SIZE (512*1024) /* 512 K info for low mem*/
#else
#define MAX_HANG_INFO_SIZE (2*1024*1024) /* 2M info */
#endif

static int MaxHangInfoSize = MAX_HANG_INFO_SIZE;
#define MAX_STRING_SIZE 256
char hang_buff[MAX_HANG_INFO_SIZE];
```

4、 以及buffer执行机制？

Buffer满后，丢掉需要添加的信息

```
static void Log2HangInfo(const char *fmt, ...)
{
        ...
        if ((Hang_Info_Size + MAX_STRING_SIZE) >=
                                (unsigned long)MaxHangInfoSize)
                return;
        ....
}
```

5、如何判断buffer 是否溢出？

查看keyword 出现的次数：” dump backtrace start”

SYS_HANG_DETECT_RAW：

Line 2: dump backtrace start: 2251787086089

Line 8629: dump backtrace start: 2282540542700

如果打印两次，则说明第一次打印是完整的。

小结：

在SYS_HANG_DETECT_RAW 未溢出的情况下，未找到对应的进程，则有可能未处于 TASK_RUNNING， TASK_UNINTERRUPTIBLE

Case 2： Kernel suspend 流程卡死, 导致hang detect 重启

现象：从后台返回，出现比较多的hang detect thread KE

看到大批量的thread 全部卡住，并且watchdog 的状态是

```
[179942.389198] 0)watchdog       D
[179942.389206] 0) ffffffc000085a84
[179942.389207] 0)    0  1374    320 0x00000008
[179942.389210] 0)Call trace:
[179942.389221] 0)[] __switch_to+0x74/0x8c
[179942.389234] 0)[] __schedule+0x314/0x794
[179942.389248] 0)[] schedule+0x24/0x68
[179942.389258] 0)[] __refrigerator+0x6c/0xf4
[179942.389269] 0)[] futex_wait_queue_me+0x150/0x158
[179942.389280] 0)[] futex_wait+0x120/0x20c
[179942.389291] 0)[] do_futex+0x184/0xa48
[179942.389301] 0)[] SyS_futex+0x88/0x19c
[179942.389313] 0)android.bg       D
[179942.389320] 0) ffffffc000085a84
[179942.389322] 0)    0  1447    320 0x00000008
[179942.389325] 0)Call trace:
[179942.389336] 0)[] __switch_to+0x74/0x8c
[179942.389350] 0)[] __schedule+0x314/0x794
[179942.389363] 0)[] schedule+0x24/0x68
[179942.389372] 0)[] __refrigerator+0x6c/0xf4
[179942.389385] 0)[] do_nanosleep+0x108/0x110
[179942.389396] 0)[] hrtimer_nanosleep+0x8c/0x108
[179942.389407] 0)[] SyS_nanosleep+0x90/0xa8
```

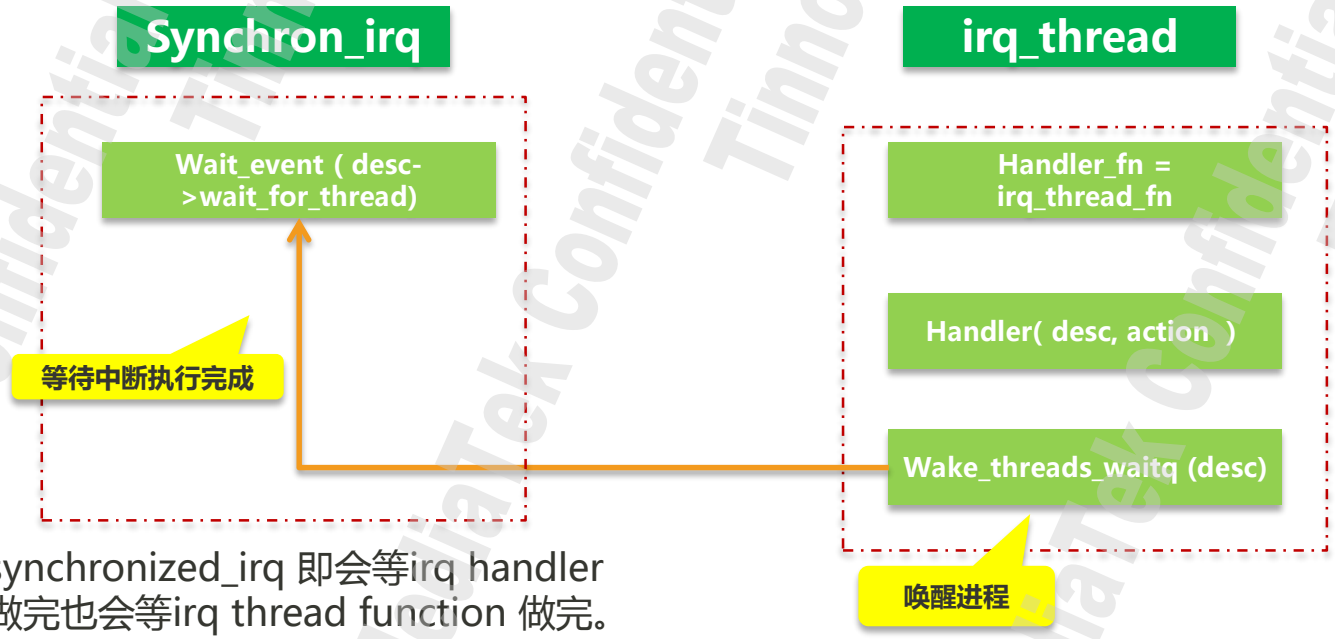
会发现很特别的__refrigerator 这个函数，怎么会停留在这里呢，貌似有点不占边， 其实此是当机器suspend 时，freezer user space task 后，强行将process 切换到__refrigerator 中等待

如果你看到这样的情况，那么即说明当时系统的suspend 流程卡住了，并且不是卡在了driver 的early suspend 流程，而是pm_suspend 流程中了，因此你需要快速的找到suspend 的kworker 卡在了哪里.

```
[179942.393792] 0)Workqueue: autosleep try_to_suspend
[179942.393795] 0)Call trace:
[179942.393807] 0)[] __switch_to+0x74/0x8c
[179942.393821] 0)[] __schedule+0x314/0x794
[179942.393835] 0)[] schedule+0x24/0x68
[179942.393850] 0)[] synchronize_irq+0x88/0xc4
[179942.393865] 0)[] suspend_device_irqs+0xdc/0xe4
[179942.393878] 0)[] dpm_suspend_noirq+0x2c/0x288
[179942.393889] 0)[] dpm_suspend_end+0x34/0x84
[179942.393905] 0)[] suspend_devices_and_enter+0x150/0x4a4
[179942.393917] 0)[] enter_state+0x15c/0x190
[179942.393929] 0)[] pm_suspend+0x2c/0xa8
[179942.393939] 0)[] try_to_suspend+0x148/0x1a8
[179942.393955] 0)[] process_one_work+0x148/0x468
[179942.393968] 0)[] worker_thread+0x138/0x3c0
[179942.393979] 0)[] kthread+0xb0/0xbc
```

在这个过程中，有执行synchronize_irq， 这个是等目前的IRQ做完为后续的disable irq: arch_suspend_disable_irqs 做准备.

当时的第一直觉是，这个irq handler 执行太久，按理早就WDT重启了，不至于等到hang detect 来检测到了。 于是看了一下synchronize_irq 的代码.



synchronized_irq 即会等irq handler 做完也会等irq thread function 做完。

此问题就变成了肯定有人注册了irq thread function, 并且没有执行完毕，因为这个thread 肯定从irq_thread 中执行，于是追查，可以看到：

```
[179942.389537] 0)irq/291-spi0.0  D
[179942.389544] 0) ffffffc000085a84
[179942.389546] 0)    0  1090      2 0x00000000
[179942.389549] 0)Call trace:
[179942.389561] 0)[] __switch_to+0x74/0x8c
[179942.389575] 0)[] __schedule+0x314/0x794
[179942.389588] 0)[] schedule+0x24/0x68
[179942.389601] 0)[] schedule_timeout+0x128/0x20c
[179942.389614] 0)[] __down_timeout+0x68/0xc8
[179942.389625] 0)[] down_timeout+0x5c/0x84
[179942.389640] 0)[] connection_read_data+0x38/0xac
[179942.389654] 0)[] mc_wait_notification+0xd0/0x148
[179942.389670] 0)[] gf516m_send_cmd_secdrv+0x68/0x1e8
[179942.389681] 0)[] gf516m_irq+0x50/0x638
[179942.389692] 0)[] irq_thread+0x110/0x164
[179942.389703] 0)[] kthread+0xb0/0xbc
```

于是这个问题就变得很明显了，gf516m 这个指纹识别的irq, 通过request_thread_irq 注册，然后和TEE 通讯mc_wait_notification, 结果卡住了，从而导致hang detect 重启.

找汇顶的工程师协助处理，将流程改成irq — work queue 的方式执行，以规避掉目前的情况。

everyday genius