

# Hexo

2017-10-18

## iOS内存abort(Jetsam) 原理探究

### 招人

手淘架构组招人 iOS / Android 皆可，地点杭州，有兴趣的请联系我！！

### iOS内存abort(Jetsam) 原理探究

苹果最近开源了iOS系统上的XNU内核代码，加上最近又开始负责手淘/猫客的稳定性及性能相关的工作，所以赶紧拜读下苹果的大作。今天主要开始想分析跟abort相关的内存Jetsam原理。

### 什么是Jetsam

关于Jetsam，可能有些人还不是很理解。我们可以从手机设置->隐私->分析这条路径看看系统的日志，会发现手机上有许多 JetsamEvent 开头的日志。打开这些日志，一般会显示一些内存大小，CPU时间什么的数据。

之所以会发生这么JetsamEvent，主要还是由于iOS设备不存在交换区导致的内存受限，所以iOS内核不得不把一些优先级不高或者占用内存过大的杀掉。这些 JetsamEvent 就是系统在杀掉App后记录的一些数据信息。

从某种程度来说，JetsamEvent是一种另类的Crash事件，但是在常规的Crash捕获工具中，由于iOS上能捕获的信号量的限制，所以因为内存导致App被杀掉是无法被捕获的。为此，许多业界的前辈通过设计 flag 的方式自己记录所谓的 abort 事件来采集数据。但是这种采集的abort，一般情况下都只能简单的记录次数，而没有详细的堆栈。

### 源码探究

MacOS/iOS是一个从BSD衍生而来的系统。其内核是Mach，但是对于上层暴露的接口一般都是基于BSD层对于Mach包装后的。虽然说Mach是个微内核的架构，真正的虚拟内存管理是在其中进行，但是BSD对于内存管理提供了相对较为上层的接口，同时，各种常见的**JetSam**事件也是由**BSD**产生，所以，我们从 `bsd_init` 这个函数作为入口，来探究下原理。

`bsd_init` 中基本都是在初始化各个子系统，比如虚拟内存管理等等。

跟内存相关的包括如下几步可能：

1. 初始化BSD内存Zone，这个Zone是基于Mach内核的zone构建 `kmeminit()`;

2. iOS上独有的特性，内存和进程的休眠的常驻监控线程

```
#if CONFIG_FREEZE
#ifdef CONFIG_MEMORYSTATUS
    #error "CONFIG_FREEZE defined without matching CONFIG_MEMORYSTATUS"
#endif
    /* Initialise background freezing */
    bsd_init_kprintf("calling memormystatus_freeze_init\n");
    memormystatus_freeze_init();
#endif>
```

3. iOS独有，JetSAM（即低内存事件的常驻监控线程）

```
#if CONFIG_MEMORYSTATUS
    /* Initialize kernel memory status notifications */
    bsd_init_kprintf("calling memormystatus_init\n");
    memormystatus_init();
#endif /* CONFIG_MEMORYSTATUS */
```

这两步代码都是调用 `kern_memormystatus.c` 里面暴露的接口，主要的作用就是从内核中开启了两个最高优先级的线程，来监控整个系统的内存情况。

首先先来看看 `CONFIG_FREEZE` 涉及的功能。当启用这个效果的时候，内核会对**进程**进行冷冻而不是Kill。

这个冷冻的功能是通过在内核中启动一个 `memormystatus_freeze_thread` 进行。这个线程在收到信号后调用 `memormystatus_freeze_top_process` 进行冷冻。

当然，涉及到进程休眠相关的代码，就需要谈谈苹果系统里面其他相关概念了。扯开又是一个比较大的话题，后续单独开文章来进行阐述。

回到iOS Abort问题上的话，我们只需要关注 `memormystatus_init` 即可，去除平台无关的代码后如下：

```

__private_extern__ void
memorystatus_init(void)
{
    thread_t thread = THREAD_NULL;
    kern_return_t result;
    int i;

    /* Init buckets */
    // 注意点1: 优先级数组, 每个数组都持有了一个同优先级进程的列表
    for (i = 0; i < MEMSTAT_BUCKET_COUNT; i++) {
        TAILQ_INIT(&memstat_bucket[i].list);
        memstat_bucket[i].count = 0;
    }
    memorystatus_idle_demotion_call = thread_call_allocate((thread_call_func_t)

#if CONFIG_JETSAM

    nanoseconds_to_absolutetime((uint64_t)DEFERRED_IDLE_EXIT_TIME_SECS * NSEC_I
    nanoseconds_to_absolutetime((uint64_t)DEFERRED_IDLE_EXIT_TIME_SECS * NSEC_I

    /* Apply overrides */
    // 注意点2: 获取一系列内核参数
    PE_get_default("kern.jetsam_delta", &delta_percentage, sizeof(delta_percent
    if (delta_percentage == 0) {
        delta_percentage = 5;
    }
    assert(delta_percentage < 100);
    PE_get_default("kern.jetsam_critical_threshold", &critical_threshold_perce
    assert(critical_threshold_percentage < 100);
    PE_get_default("kern.jetsam_idle_offset", &idle_offset_percentage, sizeof(
    assert(idle_offset_percentage < 100);
    PE_get_default("kern.jetsam_pressure_threshold", &pressure_threshold_perce
    assert(pressure_threshold_percentage < 100);
    PE_get_default("kern.jetsam_freeze_threshold", &freeze_threshold_percentag
    assert(freeze_threshold_percentage < 100);

    if (!PE_parse_boot_argn("jetsam_aging_policy", &jetsam_aging_policy,
        sizeof (jetsam_aging_policy))) {

        if (!PE_get_default("kern.jetsam_aging_policy", &jetsam_aging_policy,
            sizeof(jetsam_aging_policy))) {

            jetsam_aging_policy = kJetsmAgingPolicyLegacy;
        }
    }
}

```

```
if (jetsam_aging_policy > kJetsamAgingPolicyMax) {
    jetsam_aging_policy = kJetsamAgingPolicyLegacy;
}

switch (jetsam_aging_policy) {

    case kJetsamAgingPolicyNone:
        system_procs_aging_band = JETSAM_PRIORITY_IDLE;
        applications_aging_band = JETSAM_PRIORITY_IDLE;
        break;

    case kJetsamAgingPolicyLegacy:
        /*
         * Legacy behavior where some daemons get a 10s protection once
         * AND only before the first clean->dirty->clean transition before
         * going into IDLE band.
         */
        system_procs_aging_band = JETSAM_PRIORITY_AGING_BAND1;
        applications_aging_band = JETSAM_PRIORITY_IDLE;
        break;

    case kJetsamAgingPolicySysProcsReclaimedFirst:
        system_procs_aging_band = JETSAM_PRIORITY_AGING_BAND1;
        applications_aging_band = JETSAM_PRIORITY_AGING_BAND2;
        break;

    case kJetsamAgingPolicyAppsReclaimedFirst:
        system_procs_aging_band = JETSAM_PRIORITY_AGING_BAND2;
        applications_aging_band = JETSAM_PRIORITY_AGING_BAND1;
        break;

    default:
        break;
}

/*
 * The aging bands cannot overlap with the JETSAM_PRIORITY_ELEVATED_INACTIVE
 * band and must be below it in priority. This is so that we don't have to
 * our 'aging' code worry about a mix of processes, some of which need to c
 * and some others that need to stay elevated in the jetsam bands.
 */
assert(JETSAM_PRIORITY_ELEVATED_INACTIVE > system_procs_aging_band);
assert(JETSAM_PRIORITY_ELEVATED_INACTIVE > applications_aging_band);

/* Take snapshots for idle-exit kills by default? First check the boot-arg
if (!PE_parse_boot_argn("jetsam_idle_snapshot", &memorystatus_idle_snapsho
    /* ...no boot-arg, so check the device tree */
```

```

        PE_get_default("kern.jetsam_idle_snapshot", &memorystatus_idle_snap
    }

    memorystatus_delta = delta_percentage * atop_64(max_mem) / 100;
    memorystatus_available_pages_critical_idle_offset = idle_offset_percentage
    memorystatus_available_pages_critical_base = (critical_threshold_percentage
    memorystatus_policy_more_free_offset_pages = (policy_more_free_offset_perce

/* Jetsam Loop Detection */
if (max_mem <= (512 * 1024 * 1024)) {
    /* 512 MB devices */
    memorystatus_jld_eval_period_msecs = 8000;    /* 8000 msecs == 8 seconds
} else {
    /* 1GB and larger devices */
    memorystatus_jld_eval_period_msecs = 6000;    /* 6000 msecs == 6 seconds
}

memorystatus_jld_enabled = TRUE;

/* No contention at this point */
memorystatus_update_levels_locked(FALSE);

#endif /* CONFIG_JETSAM */

memorystatus_jetsam_snapshot_max = maxproc;
memorystatus_jetsam_snapshot =
    (memorystatus_jetsam_snapshot_t*)kalloc(sizeof(memorystatus_jetsam_snap
    sizeof(memorystatus_jetsam_snapshot_entry_t) * memorystatus_jetsam_snap
if (!memorystatus_jetsam_snapshot) {
    panic("Could not allocate memorystatus_jetsam_snapshot");
}

nanoseconds_to_absolutetime((uint64_t)JETSAM_SNAPSHOT_TIMEOUT_SECS * NSEC_I

memset(&memorystatus_at_boot_snapshot, 0, sizeof(memorystatus_jetsam_snapsh

result = kernel_thread_start_priority(memorystatus_thread, NULL, 95 /* MAXI
if (result == KERN_SUCCESS) {
    thread_deallocate(thread);
} else {
    panic("Could not create memorystatus_thread");
}
}
}

```

下面先介绍几个知识点

- 内核里面对于所有的进程都有一个优先级的分布，通过一个数组维护，数组每一项是一个进程的list。这个数组的大小是 JETSAM\_PRIORITY\_MAX + 1。其结构体定义如下：

---

```
typedef struct memstat_bucket {
    TAILQ_HEAD(, proc) list;
    int count;
} memstat_bucket_t;
```

---

这结构体非常通俗易懂。

- 线程在Mach下采用了不同的优先级，其中 MAXPRI\_KERNEL 代表的是分配给内核可用范围内最高优先级的线程。其他级别还有如下这些：

---

```
* // 优先级最高的实时线程（不太清楚谁用）
* 127      Reserved (real-time)
*          A
*          +
*          (32 levels)
*          +
*          V
* 96       Reserved (real-time)
* // 给内核用的线程优先级(MAXPRI_KERNEL)
* 95       Kernel mode only
*          A
*          +
*          (16 levels)
*          +
*          V
* 80       Kernel mode only
* // 给操作系统分配的线程优先级
* 79       System high priority
*          A
*          +
*          (16 levels)
*          +
*          V
* 64       System high priority
* // 剩下的全是用户态的普通程序可以用的
* 63       Elevated priorities
*          A
*          +
*          (12 levels)
*          +
*          V
```

```

* 52      Elevated priorities
* 51      Elevated priorities (incl. BSD +nice)
*          A
*          +
*          (20 levels)
*          +
*          V
* 32      Elevated priorities (incl. BSD +nice)
* 31      Default (default base for threads)
* 30      Lowered priorities (incl. BSD -nice)
*          A
*          +
*          (20 levels)
*          +
*          V
* 11      Lowered priorities (incl. BSD -nice)
* 10      Lowered priorities (aged pri's)
*          A
*          +
*          (11 levels)
*          +
*          V
* 0       Lowered priorities (aged pri's / idle)
*****

```

- 从上图不难看出，用户态的应用程序的线程不可能高于操作系统和内核。而且，在用户态的应用程序间的线程优先级分配也有区别，前台活动的应用程序优先级高于后台的应用程序。**iOS上大名鼎鼎的SpringBoard是应用程序中优先级最高的程序。**
- 当然线程的优先级也不是一成不变。Mach会针对每一个线程的利用率和整体系统负载动态调整优先级。如果耗费CPU太多就降低优先级，如果一个线程过度挨饿CPU则会提升其优先级。**但是无论怎么变，程序都不能超过其所在的线程优先级区间范围。**

好，预备知识说完，那苹果究竟是怎么处理 JetSam 事件呢？

```
result = kernel_thread_start_priority(memorystatus_thread, NULL, 95 /* MAXPRI_I
```

苹果其实处理的思路非常简单。如上述代码，BSD层起了一个内核优先级最高的线程 VM\_memorystatus，这个线程会在维护两个列表，一个是我们之前提到的基于进程优先级的进程列表，还有一个是所谓的内存快照列表，即保存了每个进程消耗的内存页 memorystatus\_jetsam\_snapshot。

这个常驻线程接受从内核对于内存的守护程序 pageout 通过内核调用给每个App进程发送的内存压力通知，来处理事件，这个事件转发成上层的UI事件就是平常我们会收到的全局内存警告或者每个 ViewController里面的 didReceiveMemoryWarning。

当然，我们自己开发的App是不会主动注册监听这个内存警告事件的，帮助我们在底层完成这一切的都是 `libdispatch`，如果你感兴趣的话，可以钻研下 `_dispatch_source_type_memorypressure` 和 `__dispatch_source_type_memorystatus`。

那么在哪些情况下会出现内存压力呢？我们来看一看 `memorystatus_action_needed` 这段函数：

```
static boolean_t
memorystatus_action_needed(void)
{
    #if CONFIG_EMBEDDED
        return (is_reason_thrashing(kill_under_pressure_cause) ||
                is_reason_zone_map_exhaustion(kill_under_pressure_cause) ||
                memorystatus_available_pages <= memorystatus_available_pages_pressur
    #else /* CONFIG_EMBEDDED */
        return (is_reason_thrashing(kill_under_pressure_cause) ||
                is_reason_zone_map_exhaustion(kill_under_pressure_cause));
    #endif /* CONFIG_EMBEDDED */
}
```

概括来说：

频繁的页面换进换出 `is_reason_thrashing`，Mach Zone耗尽了

`is_reason_zone_map_exhaustion`（这个涉及Mach内核的虚拟内存管理了，单独写）以及可用的页低于一个门槛了 `memorystatus_available_pages`。

在这几种情况下，就会准备去Kill 进程了。但是，在这个处理下面，有一段代码特别有意思，我们看看这个函数 `memorystatus_act_aggressive`：

```
if ( (jld_bucket_count == 0) ||
      (jld_now_msecs > (jld_timestamp_msecs + memorystatus_jld_eval_period_msecs)

/*
 * Refresh evaluation parameters
 */
jld_timestamp_msecs      = jld_now_msecs;
jld_idle_kill_candidates = jld_bucket_count;
*jld_idle_kills           = 0;
jld_eval_aggressive_count = 0;
jld_priority_band_max    = JETSAM_PRIORITY_UI_SUPPORT;
}
```

这段代码很明显，是基于某个时间间隔在做条件判断。如果不满足这个判断，后续真正执行的Kill也不会走到。那我们来看看 `memorystatus_jld_eval_period_msecs` 这个变量：



```

/* Jetsam Loop Detection */
if (max_mem <= (512 * 1024 * 1024)) {
    /* 512 MB devices */
    memorystatus_jld_eval_period_msecs = 8000;    /* 8000 msecs == 8 second window */
} else {
    /* 1GB and larger devices */
    memorystatus_jld_eval_period_msecs = 6000;    /* 6000 msecs == 6 second window */
}

```

这个时间窗口是根据设备的物理内存上限来设定的，但是无论如何，看起来至少有个**6秒**的时间可以给我们来做点事情。

当然，如果满足了时间窗口的需求，就会根据我们提到的优先级进程列表进行寻找可杀目标：

```

proc_list_lock();
switch (jetsam_aging_policy) {
case kJetsamAgingPolicyLegacy:
    bucket = &memstat_bucket[JETSAM_PRIORITY_IDLE];
    jld_bucket_count = bucket->count;
    bucket = &memstat_bucket[JETSAM_PRIORITY_AGING_BAND1];
    jld_bucket_count += bucket->count;
    break;
case kJetsamAgingPolicySysProcsReclaimedFirst:
case kJetsamAgingPolicyAppsReclaimedFirst:
    bucket = &memstat_bucket[JETSAM_PRIORITY_IDLE];
    jld_bucket_count = bucket->count;
    bucket = &memstat_bucket[system_procs_aging_band];
    jld_bucket_count += bucket->count;
    bucket = &memstat_bucket[applications_aging_band];
    jld_bucket_count += bucket->count;
    break;
case kJetsamAgingPolicyNone:
default:
    bucket = &memstat_bucket[JETSAM_PRIORITY_IDLE];
    jld_bucket_count = bucket->count;
    break;
}

bucket = &memstat_bucket[JETSAM_PRIORITY_ELEVATED_INACTIVE];
elevated_bucket_count = bucket->count;

```

**需要注意的是，JETSAM不一定只杀一个进程，他可能会大杀特杀，杀掉N多进程。**

```
if (memorystatus_avail_pages_below_pressure()) {  
    /*  
     * Still under pressure.  
     * Find another pinned processes.  
     */  
    continue;  
} else {  
    return TRUE;  
}
```

至于杀进程的话，最终都会落到函数 `memorystatus_do_kill` -> `jetsam_do_kill` 去执行。

## 其他

看苹果代码的时候，发现了不少内核的参数，一一进行了尝试后，发现 `sysctlname` 和 `sysctl` 的系统调用都被苹果禁用了，比如这些：

```
"kern.jetsam_delta"  
"kern.jetsam_critical_threshold"  
"kern.jetsam_idle_offset"  
"kern.jetsam_pressure_threshold"  
"kern.jetsam_freeze_threshold"  
"kern.jetsam_aging_policy"
```

不过，我试了下通过 `kern.boottime` 获取机器的开机时间还是可以的，代码示例如下：

```
size_t size;  
sysctlbyname("kern.boottime", NULL, &size, NULL, 0);  
  
char *boot_time = malloc(size);  
sysctlbyname("kern.boottime", boot_time, &size, NULL, 0);  
  
uint32_t timestamp = 0;  
memcpy(&timestamp, boot_time, sizeof(uint32_t));  
free(boot_time);  
  
NSDate* bootTime = [NSDate dateWithTimeIntervalSince1970:timestamp];
```

## 最后

嘻嘻，技术原理研究了一些，心里顿时对解决公司的Abort问题有了一定的眉目。嘿嘿，我写了个DEMO验证了我的思路，是可行的。哇咔咔。等我的好消息吧～

#XNU #iOS

Comments

Share

NEWER

注意系统库的坑之load函数调用多次

OLDER

基于桥的全量方法Hook方案 - 探究苹果主线程检查实现

0条评论


satanwoo

1 登录

推荐

分享

评分最高



开始讨论...

通过以下方式登录

或注册一个 DISQUS 帐号 ?


姓名

来做第一个留言的人吧！

在 SATANWOO 上还有

微信高性能线上日志系统xlog剖析

5条评论 • 4个月前



everettjf — 膜拜。整体架构图如下： 昨下面木有了呢

注意系统库的坑之load函数调用多次

1条评论 • 1个月前



everettjf — 给力给力给力

基于桥的全量方法Hook方案 - 探究苹果主线程检查实现

5条评论 • 3个月前



杨萧玉 — 太强了

从Immutable来谈谈对于线程安全的理解误区

4条评论 • 7个月前



Dexter — 好的，感谢

订阅

在您的网站上使用 Disqus添加 Disqus添加

隐私

## TAGS

[Android](#) (1)  
[C](#) (1)  
[Growth](#) (1)  
[JavaScript](#) (3)  
[Math](#) (1)  
[Performance](#) (1)  
[R](#) (2)  
[Reverse Engineering](#) (6)  
[Swift](#) (6)  
[XNU](#) (1)  
[iOS](#) (32)  
[shell](#) (1)

## TAG CLOUD

[Android](#) [C](#) [Growth](#) [JavaScript](#) [Math](#) [Performance](#) [R](#) [Reverse Engineering](#) [Swift](#) [XNU](#) [iOS](#) [shell](#)

## ARCHIVES

[November 2017](#) (2)  
[October 2017](#) (1)  
[September 2017](#) (2)  
[August 2017](#) (1)  
[July 2017](#) (1)  
[June 2017](#) (3)  
[April 2017](#) (2)  
[January 2017](#) (2)  
[October 2016](#) (1)  
[September 2016](#) (1)  
[July 2016](#) (1)  
[May 2016](#) (1)  
[April 2016](#) (2)  
[March 2016](#) (4)  
[February 2016](#) (5)  
[December 2015](#) (6)  
[November 2015](#) (5)  
[October 2015](#) (4)  
[September 2015](#) (4)

## RECENTS

[一种基于KVO的页面加载，渲染耗时监控方法](#)

注意系统库的坑之load函数调用多次

iOS内存abort(Jetsam) 原理探究

基于桥的全量方法Hook方案 - 探究苹果主线程检查实现

KVO在不同的二进制中多个符号并存的Crash问题

---

© 2017 John Doe

Powered by [Hexo](#)