Suspend and resume sequence

想了很久，还是觉得按照Android 和Linux的suspend的顺序来写会比较容易一点。

在Android上面，为了使我们的设备在使用的过程中更省电，在原版的Linux的suspend和resume流程的基础上增加了新的机制来达到省电的目标。这就是大家熟知的wake lock。

拉，先从JNI层往下，简单的介绍一下：

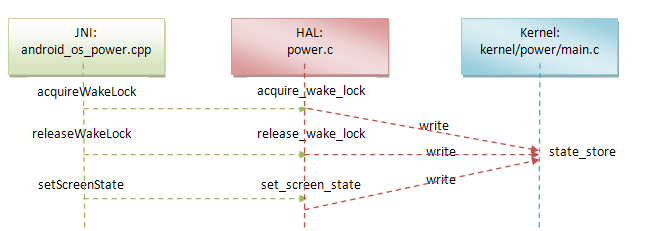


图1.0

对于framework来说，与android suspend相关的就这三个要紧的function。为了看懂这个小图补充一点背景知识，已经知道的童鞋请直接跳过。

其实，android仍然是利用了标准linux的休眠唤醒系统，只不过添加了一些使用上的新特性，early suspend、late resume、wake lock。

Early suspend - 这个机制定义了在suspend的早期，关闭显示屏的时候，一些和显示屏相关的设备，比如背光、重力感应器和触摸屏等设备都应该被关掉，但是此时系统可能还有持有wake lock的任务在运行，如音乐播放，电话，或者扫描sd卡上的文件等，这个时候整个系统还不能进入真正睡眠，直到所有的wake lock都被释放。在嵌入式设备中，电源消耗是一个很大的问题，所以android加入了这种机制。

Late resume - 这个机制定义了在resume的后期，也就是唤醒源已经将处理器唤醒，标准linux的唤醒流程已经走完了，在android上层系统识别出这个物理上的唤醒源是上层定义的，那么上层将会发出late resume的命令给下层，这个时候将会调用相关设备注册的late resume回调函数。

Wake lock - wakelock在android的电源管理系统中扮演一个核心的角色，wakelock是一种锁的机制, 只要有task拿着这个锁, 系统就无法进入休眠, 可以被用户进程和内核线程获得。这个锁可以是有超时的或者是没有超时的, 超时的锁会在时间过去以后自动解锁。如果没有锁了或者全部超时了, 内核就会启动标准linux的那套休眠机制来进入休眠。

Wake Lock的Init和Exit分别被放在core\_initcall和module\_exit的时候执行，也就是说，系统起来的时候，wake lock就被初始化好了。最后最为补充，我会介绍一下wake lock的init里面到底做了什么。

背景知识补充完毕！

然后来看看那个小图，为了看起来比较专业，将其标注为图1.0。咱们可以用自己手机看看其在suspend和resume的时候分别是什么状态，打开log，让device不停的suspend和resume，在log里面能清晰的看到set\_screen\_state 0/1这种信息的出现。其实就是在device suspend的时候set\_screen\_state 0，反之set\_screen\_state 1.当然，这只是很简单的从HAL里面出来的log，这样来看的话，个人觉得不会很绕。

其实在set\_screen\_state函数的实现是相当简单的，他就是去打开/sys/power/下面的一些节点，然后往里面写我们的”mem”或者”on”.然后就进入kernel去让device进入这个”on”或者”men”的状态。由于那张小图还没讲完，咱们先把小图说完了，在说kernel里面的实现。

那张小图上还有acquire和release wake lock，这2个function是在什么时候用的呢？举个很简单的例子：

PowerManager pm = (PowerManager)getSystemService(Context.POWER\_SERVICE);

PowerManager.WakeLock wl = pm.newWakeLock(PowerManager.SCREEN\_DIM\_WAKE\_LOCK, “Tag”);

wl.acquire(); //申请锁这个里面会调用PowerManagerService里面acquireWakeLock()

…

wl.release(); //释放锁，显示的释放，如果申请的锁不在此释放系统就不会进入休眠。

这就是一个很简单的应用实例（当然，哪些应用需要申请wake lock，估计是应用自己说了算）。这个是调用的java native里面的声明，然后才会调到我们小图上的JNI🡪HAL.

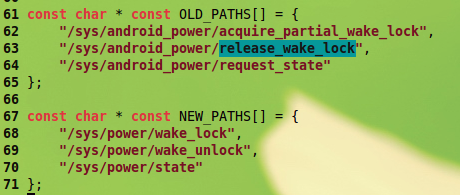
然后在来看看那个write，他们三个都调用的是write，但是明显是要实现不同的功能。所以这里就把三个拉到一起看看。

acquire wake lock: write(g\_fds[ACQUIRE\_PARTIAL\_WAKE\_LOCK], id, strlen(id)); // g\_fds[0]

release wake lock: write(g\_fds[RELEASE\_WAKE\_LOCK], id, strlen(id)); // g\_fds[1]

set screen state: write(g\_fds[REQUEST\_STATE], buf, len); // g\_fds[2]

其实这个g\_fds数组，就是前面我们说的打开的/sys/power/下面的节点，我copy过来看看：



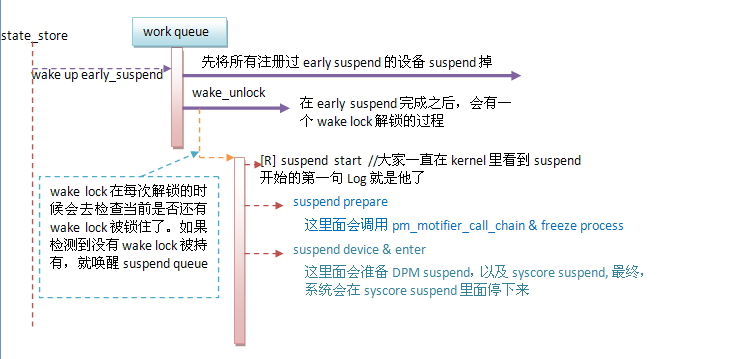
在我们现在的device上面 ，打开的是这个new patch下面的节点们。他其实就是分别去写这三个节点。这里打开的fd[0], fd[1], fd[2]就能和上面的acquire wake lock, release wake lock, set screen state相对应起来了。

到目前为止，我就当我把framework里面的一点点东西讲完了，其实framework里面本身东西没有特别多，代码量也比较小。有兴趣的话可以自己看看。

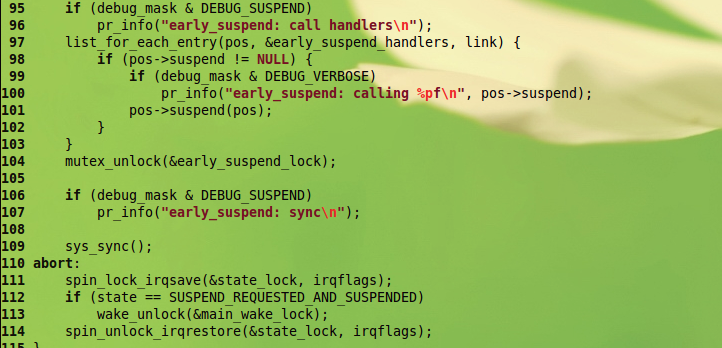
接下来是将kernel里面相关的。说起kernel里面，相信凡是拿到这个case的人都会先去看代码，然后我就不一一列举函数的call顺序了，这个大家可以自己去看看。我就把我自己总结出来的suspend的流程拿出来分享一下，至于resume，他和suspend是反过来的，所以我就只说suspend了。

其实在我们的device上面，由于android的设计，framework层下下来一个set sareen state 0的命令，让我们系统休眠。下面都是会先走early suspend的，然后才走标准的linux suspend流程（我晓得这相当于废话），问题就是，这个流程到底是怎么走完的呢？然后又是在哪里告诉system，我应该走标准的linux suspend的流程了呢？

先请看下图：



这个图有点简洁，图中有2个work queue，一个是在进state store的时候就唤醒的early suspend，另一个是如果当前没有wake lock被持有才唤醒的suspend queue。Early suspend里面是比较简单的，只要根据当前的状态，挨个调用其他设备注册的early suspend函数就好了，像是panel的early suspend函数就是在这里被调用的。



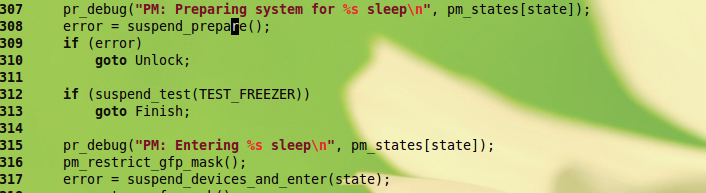
在early suspend的所有回调函数执行完成之后，会去释放main wake lock(这个wake lock是在最早的wake lock init的时候就被锁住了)。当然，这个时候wake lock被释放之后就会去检查当前是否还有wake lock被持有（其实这个检查的机制很方便的，不冗余陈述）。如果检查到wake lock没有被任何应用持有，就唤醒suspend work queue。这下就开始执行标准的linux suspend 函数了，这个时候大家就会看到一句超级标准的log”[R] suspend start”(当然，这是在咱们的QCOM平台上的啦，STE上面还没加这句话☺).然后，suspend才真正开始。

其实这个suspend整个函数真的不长，遛出来看看：



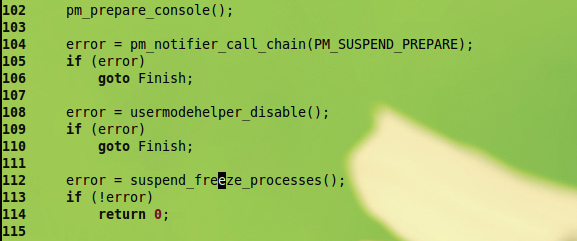
就把suspend看成一个函数，这个函数的核心就是pm\_suspend。在这里面其实就是进入了标准的linux suspend函数enter\_state（如果我们在config里面不定义config earlysuspend和config suspend 2个宏的话，我们在进入suspend的那个入口就走到这个函数了,不会进去early suspend了）。在enter state里面，我们看到他其实也只做了3件事情，第一件是sync file system. 第二件是suspend prepare，第三件是suspend device and enter。

sys_sync.png



当咱们做完这3件事之后，其实device已经进入suspend了。从函数的名字上很好理解者三件事是干什么的。 Sync file system不解释，他就是把cache该清掉的清掉，该存起来的存起来。其实我没有找到他的实现，也还没想到他有神马高级的功能，先就当他只做了这些事了。

第二件事就是suspend prepare，这个prepare顾名思义，就是准备系统进入suspend。

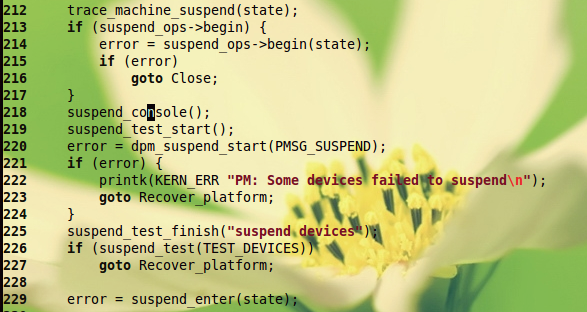


先说说pm\_prepare\_console这里面干了什么呢？事实证明，我只晓得他里面做了些跟virtual terminal相关的切换。” Perform a kernel triggered VT switch for suspend/resume”这是他的解释，有能读出深层意思的话，麻烦指点小女子一下☹.他这里面有个SUSPEND\_CONSOLE，这个值是62，貌似是在suspend情况下的一种状态还是神马的，这里还没得很清楚。

接下来是pm\_notifier\_call\_chain,根据他的调用过程显示，其实他做了三件不大事情：第一就是拿到当前的时间，这个时间后面有人做减法的时候会用的（好像就是计算一个时间差，写到一个节点里面去，我还没查出来是哪个节点，这里待定☹，好吧，我承认我还有蛮多地方没看懂的(>\_<)）,第二件事就是将所有的CPU suspend的标记位置成1，最有就打会印一句log出来。在高通平台上是酱紫的，至于STE上面，我还没去验证那个chain里面是不是也是做这三件事情，这个等有STE的device之后就可以判断了.

在接下来就是我们一直看到有印log出来的freeze processes了。他先是freeze thread，然后freeze work queue。也是按照thread和work queue的list一个一个拉出来freeze掉的。这个其实蛮要时间的，我看到的是他一般会持续一会，但是具体几秒我忘了。

Suspend prepare差不多到这里就完了，接下来就是suspend device and enter了。



这里面我也说比较要紧的三个地方。第一个是suspend console，看他的说法就是” suspend the console subsystem “, that’s means “disables printk() while we go into suspend states”. 比较明显了吧？

然后就是做了超级多事情的dpm\_suspend\_start. 我怀疑这个dpm就是动态电源管理，不过我没找到足够的证据，欢迎指证。这里面事情真的蛮多的，反正看code的话看起来是很多，但是总结成一句话的话，他其实就是讲所有注册了dpm suspend的外设全部suspend起来都是调用它们注册的回调函数，这些回调函数应该是在板级定义中定义和注册的。等这些回调函数统统走完之后，外设就应该挂起来了。然后就会进系统成的suspend，也就是下面的的suspend enter里面干的活。

在suspend enter里面，从code来看貌似干的活不太多，其实也是超级多的，全部通过回调函数来实现的, ” enter the desired system sleep state”. 啦啦，系统就是在这里面真正睡下去的。这里面除了会suspend IRQ之外，最主要的是进了syscore suspend，他会去check wakeup 的irqs，suspend VFP，suspend CPU frequency这些全部都是和power相关的，我没有去研究suspend VFP或者suspend CPU frequency里面到底做了什么，后面有机会的话在自己研究一下☹ .最后就是进入咱们各个平台不一样的地方了，高通上面叫msm\_pm\_enter，STE上面叫ux500\_suspend\_enter，整个系统都是在这里面睡下去的。我还没看懂STE的那个里面做了什么，代码量不大，不过还没看习惯☺要先熟悉一下STE平台上的那些简称，反正蛮奇怪的☹,代码在arcc/arm/mach-ux500/pm/suspend.c里面。希望有兴趣的同志能与我共勉之。

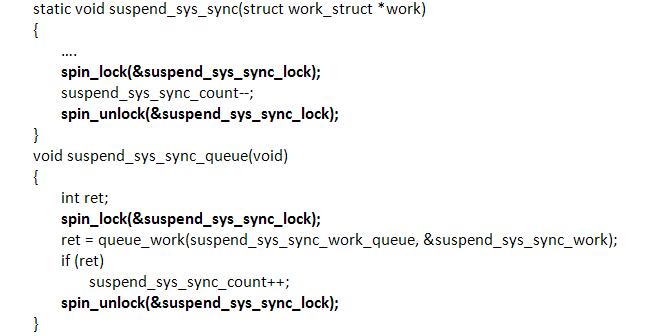
其实到了这里系统真的就已经根据我们前面那个set screen state 0 suspend了。然后其实resume就是从这里原路返回，真的是原路返回的。我就不在原路返回去在说一遍了。

知识普及完了，跟大家分享两个case，我觉得是满神奇的2个case的哇。

第一个就是大家都知道的suspend/resume很多次后会fail，该问题重现几率极小。每次出现之后除非拔电池，否则不能恢复。他看起来像是suspend之后再也resume不起来了，但是实际上是，咱们suspend的时候就已经fail了，所以系统不晓得咱们在resume，所以就是传说中的resume不起来。为什么这个东东会suspend不下去了？反复研究suspend的流程，咱们的一个很彪悍的同事提出了2个疑点（最后问题还真是出现在这里面）。

疑点1：Fs 同步，加入work queue执行，有锁

Workqueue中的函数里面有个锁，这个锁在加入时也需要用：

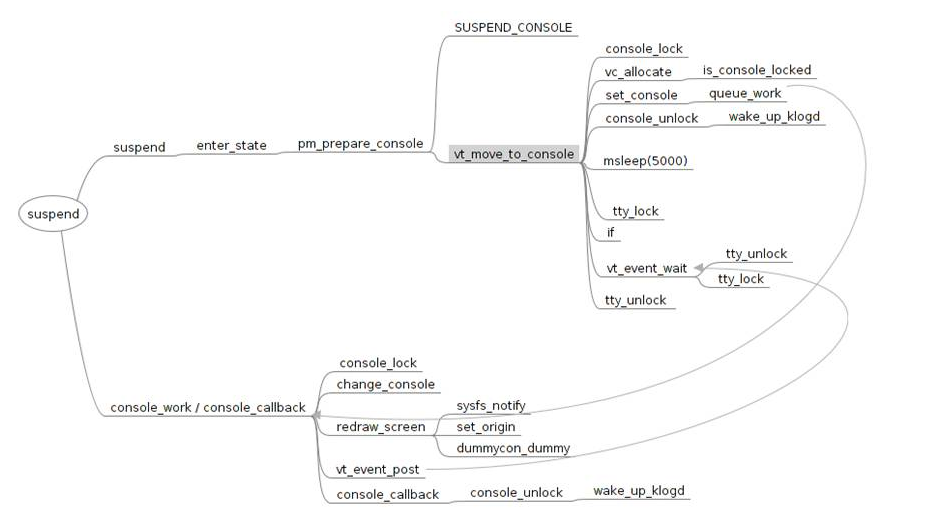


这里如果在第一个点那两个上锁解锁之间，进入到suspend，那么suspend流程里面再去调的话，就spin\_lock死锁住了。

疑点2：出现了全国人民都没人用过的pm\_prepare\_console()，并且人家的project中是没有出现这个问题的。

在notifier\_call之前有个pm\_prepare\_console()**，**这个pm\_prepare\_console()在其他机器上是不做事情的，在出问题的这个项目上开了一个宏(CONFIG\_VT\_CONSOLE)，所以…

显然，既然有疑点，咱们就应该去检验和验证，最后，经一位超级大虾指点和验证，咱们找出了这个问题在哪里。我把出问题那个地方的贴出来，各位可以自己看一下能不能看出问题在哪里，找不到的就面壁5分钟，然后回去搜邮件去。



还有一个case，大家可以自己实验一下。就是你让系统分别挂在work queue和mutex上面，看看这时候suspend的情况和被挂着的work queue和mutex会出现什么状况。当然，还分挂在系统层还是用户空间啦。推荐大家自己去实验一下，然后咱们再share一下实验结果。

好了，经验分享完毕，欢迎各位补充☺