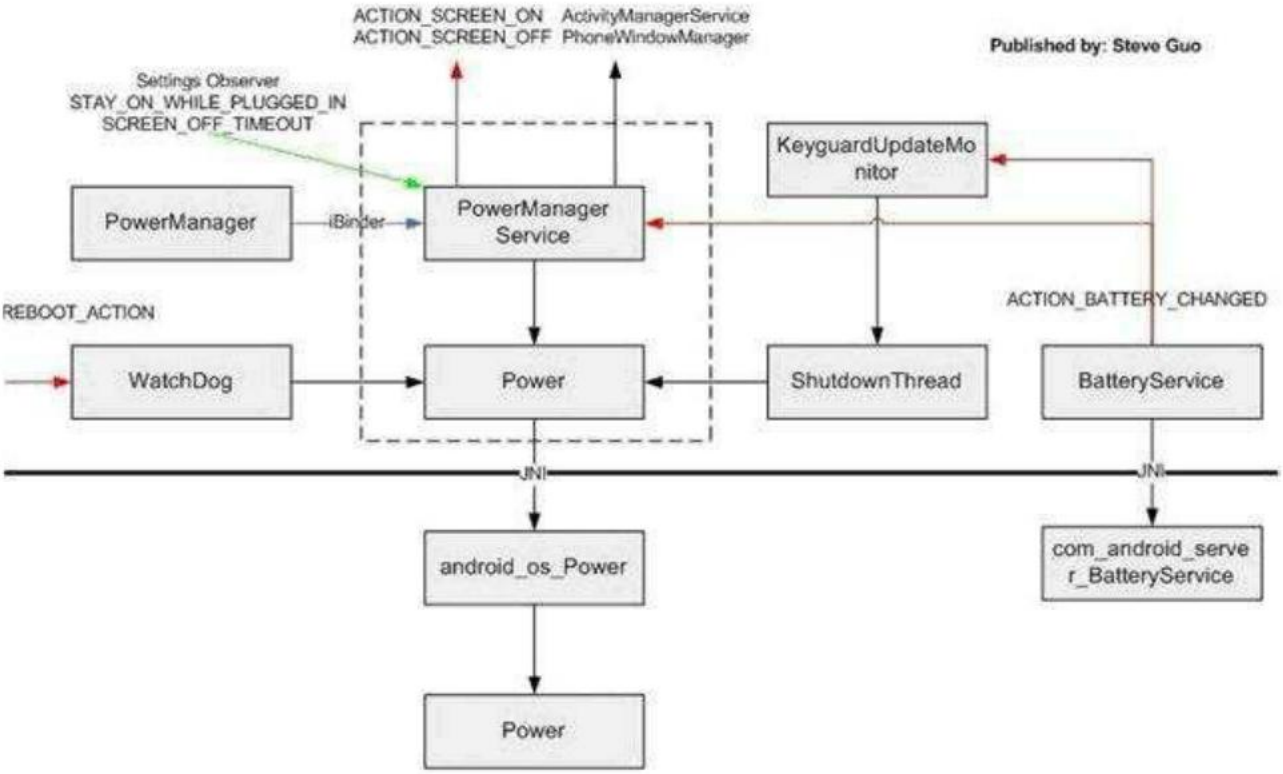
Android 的电源管理

文章分类:Java 编程

总体上来说 Android 的电源管理还是比较简单的，主要就是通过锁和定时器来切换系统的状态，使系统的功耗降至最低，整个系统的电源管理架构图如下：
(注该图来自 Steve Guo)



接下来我们从 Java 应用层面，Android framework 层面，Linux 内核层面分别进行详细的讨论：
应用层的使用：
Android 提供了现成 android.os.PowerManager 类，该类用于控制设备的电源状态的切换。
该类对外有三个接口函数：

void goToSleep(long time); // 强制设备进入 Sleep 状态

Note:
尝试在应用层调用该函数，却不能成功，出现的错误好象是权限不够，但在 Framework 下面的 Service 里调用是可以的。

newWakeLock(int flags, String tag);// 取得相应层次的锁

flags 参数说明：

PARTIAL_WAKE_LOCK: Screen off, keyboard light off

SCREEN_DIM_WAKE_LOCK: screen dim, keyboard light off

SCREEN_BRIGHT_WAKE_LOCK: screen bright, keyboard light off

FULL_WAKE_LOCK: screen bright, keyboard bright

ACQUIRE_CAUSES_WAKEUP: 一旦有请求锁时强制打开 Screen 和 keyboard light

ON_AFTER_RELEASE: 在释放锁时 reset activity timer

Note:

如果申请了 partial wakelock, 那么即使按 Power 键，系统也不会进 Sleep, 如 Music 播放时

如果申请了其它的 wakelocks, 按 Power 键，系统还是会进 Sleep

void userActivity(long when, boolean noChangeLights);//User activity 事件发生，设备会被切换到 Full on 的状态，同时 Reset Screen off timer.

Sample code:

```
PowerManager pm = (PowerManager)getSystemService(Context.POWER_SERVICE);
PowerManager.WakeLock wl = pm.newWakeLock (PowerManager.SCREEN_DIM_WAKE_LOCK, “My Tag”);
wl.acquire();
…….
wl.release();
```

Note:

1. 在使用以上函数的应用程序中，必须在其 Manifest.xml 文件中加入下面的权限：

```
<uses-permission android:name="android.permission.WAKE_LOCK" />
<uses-permission android:name="android.permission.DEVICE_POWER" />
```

2. 所有的锁必须成对的使用，如果申请了而没有及时释放会造成系统故障．如申请了 partial wakelock, 而没有及时释放，那系统就永远进不了 Sleep 模式．

Android Framework 层面：

其主要代码文件如下：

frameworks\base\core\java\android\os\PowerManager.java

frameworks\base\services\java\com\android\server\PowerManagerService.java

frameworks\base\core\java\android\os\Power.java

frameworks\base\core\jni\android_os_power.cpp

hardware\libhardware\power\power.c

其中 PowerManagerService.java 是核心，Power.java 提供底层的函数接口，与 JNI 层进行交互，JNI 层的代码主要在文件 android_os_Power.cpp 中，与 Linux kernel 交互是通过 Power.c 来实现的，Andriod 跟 Kernel 的交互主要是通过 sys 文件的方式来实现的，具体请参考 Kernel 层的介绍．

这一层的功能相对比较复杂，比如系统状态的切换，背光的调节及开关，Wake Lock 的申请和释放等等，但这一层跟硬件平台无关，而且由 Google 负责维护，问题相对会少一些，有兴趣的朋友可以自己查看相关的代码．

Kernel 层 ：
其主要代码在下列位置 ：
drivers/android/power.c
其对 Kernel 提供的接口函数有
EXPORT_SYMBOL(android_init_suspend_lock); // 初始化 Suspend lock, 在使用前必须做初始化
EXPORT_SYMBOL(android_uninit_suspend_lock); // 释放 suspend lock 相关的资源
EXPORT_SYMBOL(android_lock_suspend); // 申请 lock, 必须调用相应的 unlock 来释放它

EXPORT_SYMBOL(android_lock_suspend_auto_expire);// 申请 partial wakelock, 定时时间到后会自动释放

EXPORT_SYMBOL(android_unlock_suspend); // 释放 lock
EXPORT_SYMBOL(android_power_wakeup); // 唤醒系统到 on
EXPORT_SYMBOL(android_register_early_suspend); // 注册 early suspend 的驱动
EXPORT_SYMBOL(android_unregister_early_suspend); // 取消已经注册的 early suspend 的驱动

提供给 Android Framework 层的 proc 文件如下 ：
"/sys/android_power/acquire_partial_wake_lock" // 申请 partial wake lock
"/sys/android_power/acquire_full_wake_lock" // 申请 full wake lock
"/sys/android_power/release_wake_lock" // 释放相应的 wake lock
"/sys/android_power/request_state" // 请求改变系统状态 ，进 standby 和回到 wakeup 两种状态
"/sys/android_power/state" // 指示当前系统的状态

Android 的电源管理主要是通过 Wake lock 来实现的 ，在最底层主要是通过如下三个队列来实现其管理 ：
static LIST_HEAD(g_inactive_locks);
static LIST_HEAD(g_active_partial_wake_locks);
static LIST_HEAD(g_active_full_wake_locks);
所有初始化后的 lock 都会被插入到 g_inactive_locks 的队列中 ，而当前活动的 partial wake lock 都会被插入到 g_active_partial_wake_locks 队列中 ，活动的 full wake lock 被插入到 g_active_full_wake_locks 队列中 ，所有的 partial wake lock 和 full wake lock 在过期后或 unlock 后都会被移到 inactive 的队列 ，等待下次的调用 .
在 Kernel 层使用 wake lock 步骤如下 ：

1. 调用函数 android_init_suspend_lock 初始化一个 wake lock

2. 调用相关申请 lock 的函数 android_lock_suspend 或 android_lock_suspend_auto_expire 请求 lock, 这里只能申请 partial wake lock, 如果要申请 Full wake lock, 则需要调用函数 android_lock_partial_suspend_auto_expire(该函数没有 EXPORT 出来), 这个命名有点奇怪 ，不要跟前面的 android_lock_suspend_auto_expire 搞混了 .

3. 如果是 auto expire 的 wake lock 则可以忽略 ，不然则必须及时的把相关的 wake lock 释放掉 ，否则会造成系统长期运行在高功耗的状态 .

4. 在驱动卸载或不再使用 Wake lock 时请记住及时的调用 android_uninit_suspend_lock 释放资源 .

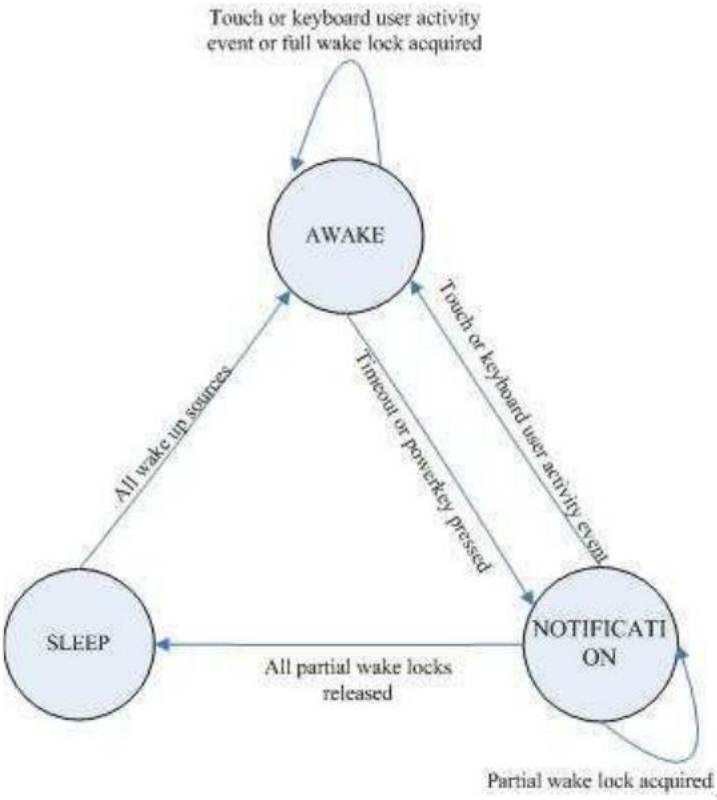
系统的状态 :

USER_AWAKE, //Full on status

USER_NOTIFICATION, //Early suspended driver but CPU keep on

USER_SLEEP // CPU enter sleep mode

其状态切换示意图如下 :



系统正常开机后进入到 AWAKE 状态 , Backlight 会从最亮慢慢调节到用户设定的亮度 , 系统 screen off timer(settings->sound & display-> Display settings -> Screen timeout) 开始计时 , 在计时时间到之前 , 如果有任何的 activity 事件发生 , 如 Touch click, keyboard pressed 等事件 , 则将 Reset screen off timer, 系统保持在 AWAKE 状态 . 如果有应用程序在这段时间内申请了 Full wake lock, 那么系统也将保持在 AWAKE 状态 , 除非用户按下 power key. 在 AWAKE 状态下如果电池电量低或者是用 AC 供电 screen off timer 时间到并且选中 Keep screen on while pluged in 选项 ,backlight 会被强制调节到 DIM 的状态 .

如果 Screen off timer 时间到并且没有 Full wake lock 或者用户按了 power key, 那么系统状态将被切换到 NOTIFICATION, 并且调用所有已经注册的 g_early_suspend_handlers 函数 , 通常会把 LCD 和 Backlight 驱动注册成 early suspend 类型 , 如有需要也可以把别的驱动注册成 early suspend, 这样就会在第一阶段被关闭 . 接下来系统会判断是否有 partial wake lock acquired, 如果有则等待其释放 , 在等待的过程中如果有 user activity 事件发生 , 系统则马上回到 AWAKE 状态 ; 如果没

有 partial wake lock acquired, 则系统会马上调用函数 pm_suspend 关闭其它相关的驱动，让 CPU 进入休眠状态 .

系统在 Sleep 状态时如果检测到任何一个 Wakeup source, 则 CPU 会从 Sleep 状态被唤醒，并且调用相关的驱动的 resume 函数，接下来马上调用前期注册的 early suspend 驱动的 resume 函数，最后系统状态回到 AWAKE 状态 . 这里有个问题就是所有注册过 early suspend 的函数在进 Suspend 的第一阶段被调用可以理解，但是在 resume 的时候 ,Linux 会先调用所有驱动的 resume 函数，而此时再调用前期注册的 early suspend 驱动的 resume 函数有什么意义呢 ? 个人觉得 android 的这个 early suspend 和 late resume 函数应该结合 Linux 下面的 suspend 和 resume 一起使用，而不是单独的使用一个队列来进行管理 .

由于本人对 Android 研究的时间还不长, 也许其中有些地方理解不正确, 甚至是错误的, 请大家谅解. 如果大家发现有疑问的地方, 有兴趣也可以一起来讨论.

电源管理可以说是移动设备中最关键的技术之一，特别是对于现代的智能手机，具有大屏幕，高频处理器，大内存，各种外设多（gps，camera，传感器），多任务操作系统，等特点，电源管理尤其显得重要，如果没有一个高效的电源管理方案，你的 smart phone 可能跑 2 小时就没电了。

Android 的电源管理技术有什么特点呢：

1. Application 并不直接控制电源

2. Application hold 电源状态的 “locks“

3. 如果没有 Application hold locks， Android 将进入掉电模式

Android 的电源管理有下面几个锁：

1. PARTIAL_WAKE_LOCK

- CPU on, screen off, keyboard off

- 不能通过电源按钮进入掉电模式

2. SCREEN_DIM_WAKE_LOCK

- CPU on, screen dim, keyboard off

3. SCREEN_BRIGHT_WAKE_LOCK

- CPU on, screen bright, keyboard off

4. FULL_WAKE_LOCK

- CPU on, screen on, keyboard bright

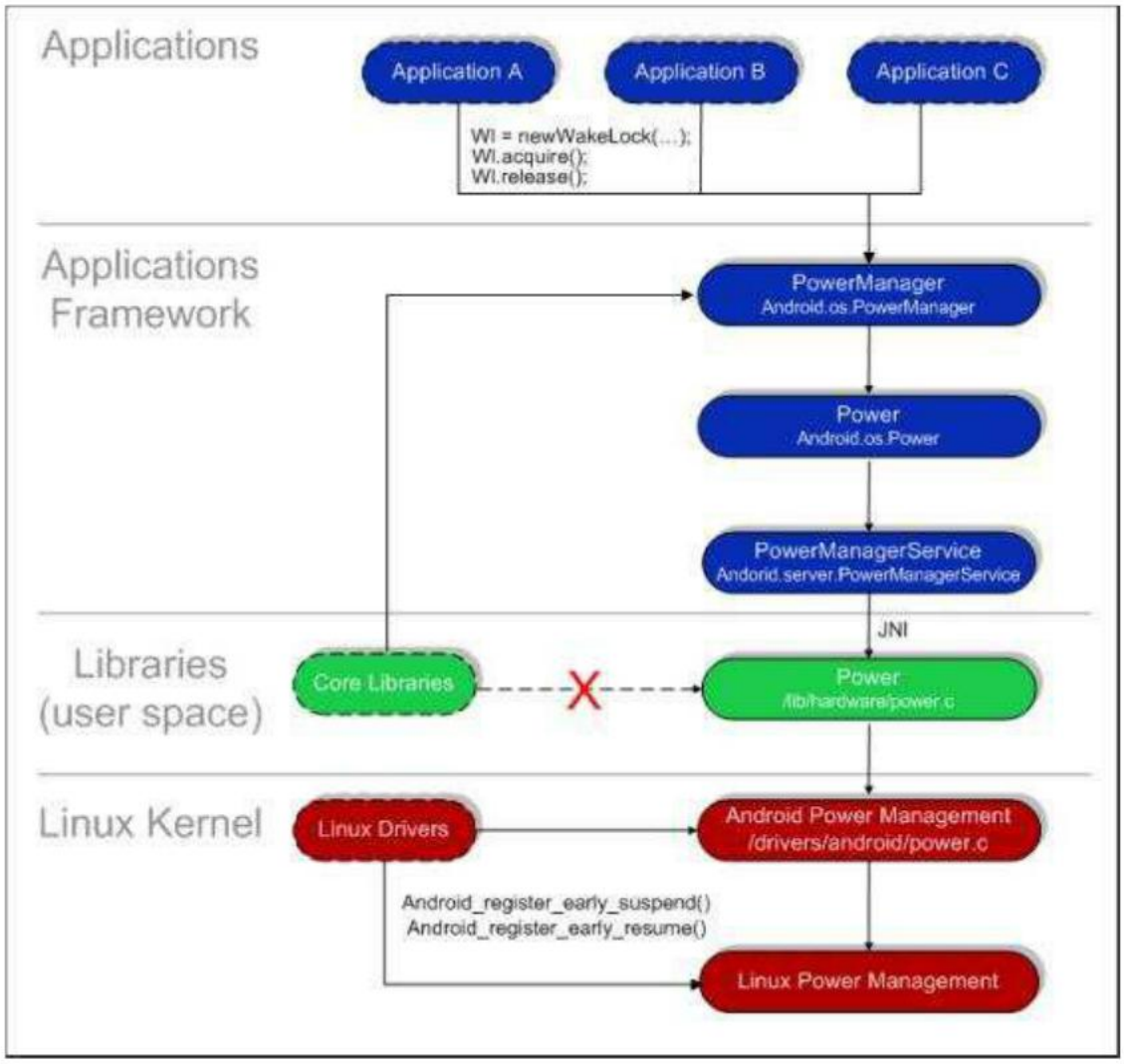
5. ACQUIRE_CAUSES_WAKEUP

- 一旦有请求锁时强制打开 Screen 和 keyboard light

6. ON_AFTER_RELEASE

- 当 lock 被释放后，通过 reset user activity timer 使屏幕多亮一会儿

Android 的电源管理构架如下图所示(图中 kernel 部分代码的路径是老版本的 android)：



下面我们看一个例子：

```
1.    PowerManager pm =
2.        (PowerManager) getSystemService(Context.POWER_SERVICE);
3.    PowerManager.WakeLock wl =
4.        pm.newWakeLock(PowerManager.SCREEN_DIM_WAKE_LOCK, "tag");
5.    wl.acquire();
6.    // ..screen will stay on during this section..
7.    wl.release();
```

这个例子首先通过 getSystemService 得到 PowerManager， 然后生成一个 SCREEN_DIM_WAKE_LOCK 锁，并且通过 acquire()方法使用这个锁，通过 release()方法释放这个锁。 acquire 和 release 必须成对使用，否则会造成系统电源管理的错误。（比如如果 acquire 了

partial_wake_lock 而忘记释放了，那么系统永远无法进入掉电模式），　还有，必须在 AndroidManifest.xml 中加入以下 permission：

```
<uses-permission android:name="android.permission.WAKE_LOCK" />

<uses-permission android:name="android.permission.DEVICE_POWER" />
```

Linux 与 Android 休眠唤醒对比（二）
2011-02-18 14:22
Android 休眠与唤醒（二）

Android 中定义了几种低功耗状态：earlysuspend，suspend，hibernation。

1，earlysuspend 是一种低功耗的状态,某些设备可以选择进入某种功耗较低的状态,比如 LCD 可以降低亮度或灭掉;

2，suspend 是指除电源管理以外的其他外围模块以及 cpu 均不工作,只有内存保持自刷新的状态;

3，hibernation 是指所有内存镜像都被写入磁盘中,然后系统关机,恢复后系统将能恢复到“关机”之前的状态.

在一个打过 android 补丁的内核中，state_store()首先判断用户写入的是否是“disk”字符串,如果是则调用 hibernate()函数命令系统进入 hibernation 状态. 如果是其他字符串则调用 request_suspend_state()（如果定义 CONFIG_EARLYSUSPEND）或者调用 enter_state()（如果未定义 CONFIG_EARLYSUSPEND）.

AndroidSuspend 时，state_store()函数会进入到 request_suspend_state()中，这个文件在 earlysuspend.c 中,这些功能都是 android 系统加的，后面会对 early suspend 和 late resume 进行介绍.

涉及到的文件:

linux_source/kernel/power/main.c

linux_source/kernel/power/earlysuspend.c

linux_source/kernel/power/wakelock.c

特性介绍

Early Suspend

Earlysuspend 是 android 引进的一种机制，这种机制在上游备受争议,这里不做评论. 这个机制作用在关闭显示的时候. 在这个时候，一些和显示有关的设备,比如 LCD 背光，重力感应器，触摸屏，这些设备都会关掉，但是系统可能还是在运行状态(这时候还有 wake lock)进行任务的处理，例如在扫描 SD 卡上的文件，后台音乐/FM 播放，文件传输/下载等. 在嵌入式设备中，背光是一个很大的电源消耗,所以 android 加入这样一种机制.

Late Resume

Late Resume 是和 Earlysuspend 配套的一种机制，是在内核唤醒完毕开始执行的. 主要就是唤醒在 Early Suspend 的时候休眠的设备.

Wake Lock

Wake Lock 在 Android 的电源管理系统中扮演一个核心的角色. Wake Lock 是一种锁的机制，只要有人拿着这个锁，系统就无法进入休眠，它可以被用户态程序和内核获得. 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. 如果没有锁了或者超时了，内核就会启动休眠的那套机制来进入休眠.

Android Suspend

当用户通过 sysfs 写入 mem 或者 standby 到/sys/power/state 中的时候，state_store()会被调用，然后 Android 会在这里调用 request_suspend_state() 而标准的 Linux 会在这里进入 enter_state()这个函数. 如果请求的是休眠，那么 early_suspend 这个 workqueue[queue_work()]就会被调用,并且进入 early_suspend 状态.

```
voidrequest_suspend_state(suspend_state_t new_state)
{
        unsignedlong irqflags;
        intold_sleep;

        spin_lock_irqsave(&state_lock,irqflags);
        old_sleep= state & SUSPEND_REQUESTED;
```

```
        if(debug_mask & DEBUG_USER_STATE) {
                structtimespec ts;
                structrtc_time tm;
                getnstimeofday(&ts);
                rtc_time_to_tm(ts.tv_sec,&tm);
                pr_info("request_suspend_state:%s (%d->%d)
at %lld "
                        "(%d-%02d-%02d%02d:%02d:%02d.%09l
u UTC)",
                        new_state!= PM_SUSPEND_ON ?
"sleep" : "wakeup",
                        requested_suspend_state,new_state
,
                        ktime_to_ns(ktime_get()),
                        tm.tm_year+ 1900, tm.tm_mon + 1,
tm.tm_mday,
                        tm.tm_hour,tm.tm_min, tm.tm_sec,
ts.tv_nsec);
        }
        if(!old_sleep && new_state != PM_SUSPEND_ON) {
                state|= SUSPEND_REQUESTED;
                queue_work(suspend_work_queue,&early_susp
end_work);
        }else if (old_sleep && new_state == PM_SUSPEND_ON)
{
                state&= ~SUSPEND_REQUESTED;
                wake_lock(&main_wake_lock);
                queue_work(suspend_work_queue,&late_resum
e_work);
        }

        requested_suspend_state= new_state;
        spin_unlock_irqrestore(&state_lock,irqflags);

}
```

EarlySuspend

在 early_suspend()函数中，首先会检查现在请求的状态还是否是 suspend，来防止 suspend 的请求会在这个时候取消掉(因为这个时候用户进程还在运行),如果需要退出，就简单的退出了．如果没有，这个函数就会把 early suspend 中注册的一系列的回调都调用一次，然后同步文件系统，然后放弃掉 main_wake_lock，这个 wake lock 是一个没有超时的锁,如果这个锁不释放，那

么系统就无法进入休眠.

```
static void early_suspend(struct work_struct *work)
{
        structearly_suspend *pos;
        unsignedlong irqflags;
        intabort = 0

        mutex_lock(&early_suspend_lock);
        spin_lock_irqsave(&state_lock,irqflags);
        if(state == SUSPEND_REQUESTED)
                state|= SUSPENDED;
        else
                abort= 1;
        spin_unlock_irqrestore(&state_lock,irqflags);

        if(abort) {
                if(debug_mask & DEBUG_SUSPEND)
                        pr_info("early_suspend:abort,
state %d", state);
                mutex_unlock(&early_suspend_lock);
                gotoabort;
        }

        if(debug_mask & DEBUG_SUSPEND)
                pr_info("early_suspend:call handlers");
        list_for_each_entry(pos,&early_suspend_handlers,
link) {
                if(pos->suspend != NULL)
                        pos->suspend(pos);
        }

        mutex_unlock(&early_suspend_lock);



        if (debug_mask & DEBUG_SUSPEND)
                pr_info("early_suspend:sync");
        sys_sync();

abort:

        spin_lock_irqsave(&state_lock,irqflags);
```

```
        if(state == SUSPEND_REQUESTED_AND_SUSPENDED)
                wake_unlock(&main_wake_lock);
        spin_unlock_irqrestore(&state_lock,irqflags);

}
```

LateResume

当所有的唤醒已经结束以后，用户进程都已经开始运行了，唤醒通常会是以下的几种原因:

来电

如果是来电，那么 Modem 会通过发送命令给 rild 来让 rild 通知 WindowManager 有来电响应,这样就会远程调用 PowerManagerService 来写"on" 到 /sys/power/state 来执行 late resume 的设备，比如点亮屏幕等.

用户按键

用户按键事件会送到 WindowManager 中,WindowManager 会处理这些 按键事件,按键分为几种情况，如果案件不是唤醒键(能够唤醒系统的按键) 那么 WindowManager 会主动放弃 wakeLock 来使系统进入再次休眠，如果按键是唤醒键,那么 WindowManger 就会调用 PowerManagerService 中的接口来执行 Late Resume. Late Resume 会依次唤醒前面调用了 Early Suspend 的设备.

```
static void late_resume(structwork_struct *work)

{

        struct early_suspend *pos;
        unsignedlong irqflags;
        intabort = 0;

        mutex_lock(&early_suspend_lock);
        spin_lock_irqsave(&state_lock,irqflags);
        if(state == SUSPENDED)
                state&= ~SUSPENDED;
        else
                abort= 1;
        spin_unlock_irqrestore(&state_lock,irqflags);
        if(abort) {
                if(debug_mask & DEBUG_SUSPEND)
                        pr_info("late_resume:abort,
```

```
state %d", state);
                gotoabort;
        }

        if(debug_mask & DEBUG_SUSPEND)
                pr_info("late_resume:call handlers");
        list_for_each_entry_reverse(pos,&early_suspend_ha
ndlers, link)
                if(pos->resume != NULL)
                        pos->resume(pos);
        if(debug_mask & DEBUG_SUSPEND)
                pr_info("late_resume:done");

abort:
        mutex_unlock(&early_suspend_lock);

}
```

Wake Lock

我们接下来看一看 wake lock 的机制是怎么运行和起作用的，主要关注 wakelock.c 文件就可以了.

wake lock 有加锁和解锁两种状态，加锁的方式有两种，一种是永久的锁住，这样的锁除非显示的放开，是不会解锁的，所以这种锁的使用是非常小心的. 第二种是超时锁，这种锁会锁定系统唤醒一段时间，如果这个时间过去了，这个锁会自动解除.

锁有两种类型:

WAKE_LOCK_SUSPEND 这种锁会防止系统进入睡眠

WAKE_LOCK_IDLE 这种锁不会影响系统的休眠，作用我不是很清楚.

在 wakelock 中，会有 3 个地方让系统直接开始 suspend()，分别是:

在 wake_unlock()中，如果发现解锁以后没有任何其他的 wake lock 了，就开始休眠 在定时器都到时间以后，定时器的回调函数会查看是否有其他的 wake lock，如果没有，就在这里让系统进入睡眠. 在 wake_lock() 中，对一个 wake lock 加锁以后，会再次检查一下有没有锁，我想这里的检查是没有必要的，更好的方法是使加锁的这个操作原子化，而 不是繁冗的检查. 而且这样的检查也有可能漏掉.

Suspend

当 wake_lock 运行 suspend()以后，在 wakelock.c 的 suspend()函数会被调用，这个函数首先 sync 文件系统,然后调用 pm_suspend(request_suspend_state)，接下来 pm_suspend()就会调用 enter_state()来进入 Linux 的休眠流程..

```
static void suspend(structwork_struct *work),
{
        intret;
        intentry_event_num;

        if(has_wake_lock(WAKE_LOCK_SUSPEND)) {
                if(debug_mask & DEBUG_SUSPEND)
                        pr_info("suspend:abort suspend");
                return;
        }

        entry_event_num= current_event_num;
        sys_sync();
        if(debug_mask & DEBUG_SUSPEND)
                pr_info("suspend:enter suspend");

        ret= pm_suspend(requested_suspend_state);

        if (current_event_num == entry_event_num) {

                wake_lock_timeout(&unknown_wakeup,HZ / 2);
        }

}
```

Android 于标准 Linux 休眠的区别

pm_suspend()虽然会调用 enter_state()来进入标准的 Linux 休眠流程,但是还是有一些区别:

1，当进入冻结进程的时候，android 首先会检查有没有 wakelock,如果没有，才会停止这些进程，因为在开始 suspend 和冻结进程期间有可能有人申请了 wake lock,如果是这样，冻结进程会被中断.

2，在 suspend_late()中，会最后检查一次有没有 wake lock，这有可能是某种快速申请 wake lock,并且快速释放这个锁的进程导致的,如果有这种情况，这里会返回错误，整个 suspend 就会全部放弃.如果 pm_suspend()成功了,LOG 的输出可以通过在 kernel cmd 里面增加"no_console_suspend" 来看到 suspend 和 resume 过程中的 log 输