
第十三章 Android 内核驱动——电源管理

13.1 基本原理

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

- 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;
- 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;
- 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。

13.2 电源管理机制的实现

电源管理机制的源代码主要在 kernel/power/文件夹下面。

main.c 文件是整个框架的入口。用户可以通过读写 sys 文件/sys/power/state 实现控制系统进入低功耗状态。用户对于/sys/power/state 的读写会调用到 main.c 中的 state_store(), 用户可以写入 const char * const pm_states[] 中定义的字符串, 比如“on”, “mem”, “standby”, “disk”。

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() (如果未定义 CONFIG_EARLYSUSPEND) 或者调用 enter_state() (如果未定义 CONFIG_EARLYSUSPEND)。

request_suspend_state()函数是 android 相对标准 linux 改动的地方, 它实现在 earlysuspend.c 中。在标准 linux 内核中, 用户通过 sysfs 写入“mem”和“standby”时, 会直接调用 enter_state() 进入 suspend 模式, 但在 android 中则会调用 request_suspend_state()函数进入 early suspend 状态。request_suspend_state()函数代码如下:

```
void request_suspend_state(suspend_state_t new_state)
{
    unsigned long irqflags;
    int old_sleep;
    spin_lock_irqsave(&state_lock, irqflags);
    old_sleep = state & SUSPEND_REQUESTED;
    .....
    if (!old_sleep && new_state != PM_SUSPEND_ON) {
        state |= SUSPEND_REQUESTED;
        //判断是否为省电请求, 如果是排队一个 early_suspend_work
        queue_work(suspend_work_queue, &early_suspend_work);
    } else if (old_sleep && new_state == PM_SUSPEND_ON) {
        state &= ~SUSPEND_REQUESTED;
        wake_lock(&main_wake_lock);
        //否则, 是唤醒请求, 排队 late_resume_work
        queue_work(suspend_work_queue, &late_resume_work);
    }
}
```

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

early_suspend_work 和 late_resume_work 定义为

```
static DECLARE_WORK(early_suspend_work, early_suspend);
static DECLARE_WORK(late_resume_work, late_resume);
```

可见实际工作的是 early_suspend 和 late_resume 这两个函数。Android 提供了 register_early_suspend 和 unregister_early_suspend 两个函数供驱动调用，分别完成设备 early_suspend 的注册和注销。系统将所有注册支持 early_suspend 的设备驱动对应的 handler 挂在一个称为 early_suspend_handler 的链表上。函数 early_suspend 和 late_resume 完成的事情很简单，就是遍历这个链表，依次调用每个设备注册的 handler，late_resume 是唤醒处于 early_suspend 的那些设备。代码如下：

```
static void early_suspend(struct work_struct *work)
{
    struct early_suspend *pos;
    unsigned long irqflags;
    int abort = 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\n", state);
        mutex_unlock(&early_suspend_lock);
        goto abort;
    }
    if (debug_mask & DEBUG_SUSPEND)
        pr_info("early_suspend: call handlers\n");

    //遍历链表依次调用每个驱动的 handler
    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\n");
    //同步文件系统
    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);
}
```

```
static void late_resume(struct work_struct *work)
{
    struct early_suspend *pos;
    unsigned long irqflags;
    int abort = 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\n", state);
        goto abort;
    }
    if (debug_mask & DEBUG_SUSPEND)
        pr_info("late_resume: call handlers\n");

    //遍历链表依次调用每个驱动注册的 resume handler
    list_for_each_entry_reverse(pos, &early_suspend_handlers, link)
        if (pos->resume != NULL)
            pos->resume(pos);

    if (debug_mask & DEBUG_SUSPEND)
        pr_info("late_resume: done\n");
abort:
    mutex_unlock(&early_suspend_lock);
}

```

`register_early_suspend` 函数完成的功能就是把驱动提供的 `earlysuspend` handler 挂到 `early_suspend_handler` 链表上。`unregister_early_suspend` 则相反，从链表上摘下 handler。

`fbearlysuspend.c` 和 `consoleearlysuspend.c` 这两个文件实现了针对 `lcd framebuffer` 的 `earlysuspend` 支持和 `console` 的 `earlysuspend` 支持。实际上这两个文件就是利用上面 `earlysuspend.c` 提供的接口注册了针对 `framebuffer` 和 `console` 的 `early suspend handler`，并提供相应的 handler 函数。

`Hibernate.c` 文件实现 `hibernation` 低功耗状态，是最彻底的低功耗模式，它把所有内存镜像都写入磁盘中，然后系统关机。该文件还在 `sysfs` 文件系统中创建了多个 `entry`，分别是 `/sys/power/disk`，`/sys/power/resume` 和 `/sys/power/image_size`，这样用户可以直接通过 `sysfs` 来控制系统进出 `hibernation` 状态。这块代码跟标准 Linux 内核没有什么区别。

Android 改动较大的另一处是增加了 `wakelock` 机制。实现在 `wakelock.c` 和 `userwakelock.c` 中。`wakelock` 可以阻止处于正常运行（`active`）或者空闲（`idle`）状态的系统进入睡眠等低功耗状态。直到所持有的 `wakelock` 全部被释放，系统才能进入睡眠等低功耗的状态。

`wakelock` 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。第二种是超时锁，这种锁会锁定系统一段时间，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，这种锁可以具有 `WAKE_LOCK_AUTO_EXPIRE` 属性，具有这种属性的锁称为超时锁(timeout)。
- `WAKE_LOCK_IDLE` 这种锁不会影响系统的 `suspend`，用于阻止系统持有锁的过程中进入 `low power` 的状态。

Android 使用两条双向链表 `active_wake_locks[WAKE_LOCK_TYPE_COUNT]` 分别保存处于 `active` 状态的 `suspend lock` 和 `idle lock`；使用一条链表 `inactive_locks` 记录所有处于 `inactive` 状态的

锁。

在系统启动的时候，会调用 `wakelocks_init` 函数来完成 `wakelock` 的初始化，但别的驱动程序也可以再单独创建自用的 `wakelock`，这里初始化的是系统默认的 `wake lock` 以及该机制依赖的功能。

`wakelocks_init` 函数做了以下事情：

- 初始化 `active_wake_locks` 链表
- 调用 `wake_lock_init` 初始化 `main_wake_lock`, `unknown_wakeup` 以及 `deleted_wake_locks`（如果 `CONFIG_WAKELOCK_STAT` 被定义）三个 `WAKE_LOCK_SUSPEND` 型的锁
- 调用 `platform_device_register` 和 `platform_driver_register` 注册平台设备和驱动。
- 创建 `suspend_work_queue` 工作队列，这会在 `wake_unlock` 解锁的时候用到。
`wake_lock_init()` 函数初始化一个锁，就是初始化表示一个 `wakelock` 的数据结构 `struct wake_lock`，并将其挂到 `inactive_locks` 链表上。
- 加锁有两个函数：`wake_lock(struct wake_lock *lock)` 和 `ake_lock_timeout(struct wake_lock *lock, long timeout)`，前者是没有指定过期时间的（除非显式调用 `wake_unlock` 否则永远锁住）；后者是有过期时间的（时间过期后，锁会解锁，即使没有显式调用 `wake_unlock`）。这两个函数内部都是通过调用 `wake_lock_internal()` 函数完成具体功能的。

`wake_lock_internal()` 函数流程：

- 判断锁的类型是否有效，即是否为 `WAKE_LOCK_SUSPEND` 或 `WAKE_LOCK_IDLE` 某一种
- 如果定义了 `CONFIG_WAKELOCK_STAT`，则更新 `struct wake_lock` 里面的用于统计锁信息的成员变量
- 将锁从 `inactive_locks` 链表上取下，加到 `active_wake_locks` 链表上。如果是超期锁则设置锁的 `flag|=WAKE_LOCK_AUTO_EXPIRE`，否则取消 `WAKE_LOCK_AUTO_EXPIRE` 标志。如果锁是 `WAKE_LOCK_SUSPEND` 型的，则继续下面的步骤。
- 对于 `WAKE_LOCK_SUSPEND` 型的锁如果它是超期锁，则调用 `has_wake_lock_locked` 函数检查所有处于活动状态的 `WAKE_LOCK_SUSPEND` 锁（即在 `active_wake_locks` 链表上的 `WAKE_LOCK_SUSPEND` 锁，或者说当前被加锁了的 `WAKE_LOCK_SUSPEND` 锁），是否有超期锁已经过期，如果有则把过期超期锁从 `active_wake_locks` 上删除，挂到 `inactive_locks` 上。同时它还检查链表上有没有非超期锁，如果有则直接返回 -1，否则它最终返回的是所有超期锁过期时间的最大值
- 如果 `has_wake_lock_locked` 函数返回的是 -1（表示当前活动锁有非超时锁）或者 0（表示所有活动锁都是超时锁，且全已经超时），则删除 `expire_timer`，并排队一个 `suspend` 工作到 `suspend_work_queue` 工作队列，最终系统会 `suspend`

```
static long has_wake_lock_locked(int type)
{
    struct wake_lock *lock, *n;
    long max_timeout = 0;
    BUG_ON(type >= WAKE_LOCK_TYPE_COUNT);
    list_for_each_entry_safe(lock, n, &active_wake_locks[type], link) {
        if (lock->flags & WAKE_LOCK_AUTO_EXPIRE) {
            long timeout = lock->expires - jiffies;
            if (timeout <= 0)
                expire_wake_lock(lock);
            else if (timeout > max_timeout)
                max_timeout = timeout;
        } else
```

```

        return -1;
    }
    return max_timeout;
}

```

expire_timer 定义为:

```
static DEFINE_TIMER(expire_timer, expire_wake_locks, 0, 0);
```

其 handler `expire_wake_locks` 是实现超时锁机制的关键，定时器的 `expire` 时间被设置为当前所有处于活动状态的 `WAKE_LOCK_SUSPEND` 锁超时值的最大值，如果没有超时锁则设置 `stop` 它。当定时器 `expire` 的时候，会在其处理函数 `expire_wake_locks` 中调用 `has_wake_lock_locked` 函数把所有过期的锁全部解锁，并排队一个 `suspend` 工作到 `suspend_work_queue` 工作队列，最终系统会 `suspend`。

```

static void expire_wake_locks(unsigned long data)
{
    long has_lock;
    unsigned long irqflags;
    if (debug_mask & DEBUG_EXPIRE)
        pr_info("expire_wake_locks: start\n");
    spin_lock_irqsave(&list_lock, irqflags);
    if (debug_mask & DEBUG_SUSPEND)
        print_active_locks(WAKE_LOCK_SUSPEND);
    has_lock = has_wake_lock_locked(WAKE_LOCK_SUSPEND);
    if (debug_mask & DEBUG_EXPIRE)
        pr_info("expire_wake_locks: done, has_lock %ld\n", has_lock);
    if (has_lock == 0)
        queue_work(suspend_work_queue, &suspend_work);
    spin_unlock_irqrestore(&list_lock, irqflags);
}

```

`suspend` 函数完成 `suspend` 系统的任务，它是 `suspend_work` 这个工作的处理函数，`suspend_work` 排队到 `suspend_work_queue` 工作队列中，最终系统会处理这个 `work`，调用其 handler 即 `suspend` 函数。该函数首先 `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 `suspend` 流程。

```

static void suspend(struct work_struct *work)
{
    int ret;
    int entry_event_num;
    if (has_wake_lock(WAKE_LOCK_SUSPEND)) {
        if (debug_mask & DEBUG_SUSPEND)
            pr_info("suspend: abort suspend\n");
        return;
    }
    entry_event_num = current_event_num;
    sys_sync();
    if (debug_mask & DEBUG_SUSPEND)
        pr_info("suspend: enter suspend\n");
    ret = pm_suspend(requested_suspend_state);
    if (debug_mask & DEBUG_EXIT_SUSPEND) {
        struct timespec ts;
        struct rtc_time tm;
        getnstimeofday(&ts);
        rtc_time_to_tm(ts.tv_sec, &tm);
        pr_info("suspend: exit suspend, ret = %d "
            " (%d-%02d-%02d %02d: %02d.%09lu UTC)\n", ret,

```

```

        tm.tm_year + 1900, tm.tm_mon + 1, tm.tm_mday,
        tm.tm_hour, tm.tm_min, tm.tm_sec, ts.tv_nsec);
    }
    if (current_event_num == entry_event_num) {
        if (debug_mask & DEBUG_SUSPEND)
            pr_info("suspend: pm_suspend returned with no event\n");
        wake_lock_timeout(&unknown_wakeup, HZ / 2);
    }
}

```

解锁由 `wake_unlock` 函数实现。该函数首先将该锁从 `active` 链表转移到 `inactive` 链表中。如果是 `WAKE_LOCK_IDLE` 锁，就结束退出了。如果是 `WAKE_LOCK_SUSPEND` 锁，则继续查看所有处于 `active` 状态并且具有自动过期属性的锁（超时锁），遍历找到最晚过期时间，然后修改 `expire_timer` 的到期时间（`expire_timer` 到期后会调用 `suspend` 函数使系统进入 `suspend` 状态）；否则，如果存在一个不具有 `auto-expire` 属性的锁（非超期锁），则会导致 `expire_timer` 被 `stop`（或者说不再处于 `active` 的工作状态）。另外，如果检查的过程中发现所有锁均处于过期状态，则直接使用 `queue_work` 启动 `suspend` 过程。

`userwakelock.c` 文件实现的是 `wakelock` 机制的 `sysfs` 接口，用户可以通过这个接口操作锁，加锁或解锁。它通过 `struct user_wake_lock` 结构体将所有的锁组织成红黑树的形式，树的根为 `user_wake_locks`。

该文件是标准的 `sysfs` 接口函数，提供了 `wake_lock_show`、`wake_lock_store`、`wake_unlock_show` 和 `wake_unlock_store` 四个函数，这样用户可以通过 `echo`、`cat` 等命令写入或读出系统中 `wake lock`。

因为 `wakelock` 在实现的过程中，默认初始化并添加一个 `suspend lock` 类型的非过期型锁 `main_wake_lock`（`wakelocks_init` 函数，`wakelock.c`）。因此，系统将始终因为 `main_wakelock` 的存在而正常运行。也就是说如果不添加新锁，将 `main_wake_lock` 解锁后，系统将进入睡眠状态。

13.3 用户接口

电源管理内核层给应用层提供的接口就是 `sysfs` 文件系统，所有的相关接口都通过 `sysfs` 实现。Android 上层 `frameworks` 也是基于 `sysfs` 做了包装，最终提供给 Android java 应用程序的是 `java` 类的形式。

Android 系统会在 `sysfs` 里面创建以 `entry`:

```

/sys/power/state
/sys/power/wake_lock
/sys/power/wake_unlock

```

```
echo mem > /sys/power/state
```

或者

```
echo standby > /sys/power/state
```

命令系统进入 `earlysuspend` 状态，那些注册了 `early suspend handler` 的驱动将依次进入各自的 `earlysuspend` 状态。

```
echo on > /sys/power/state
```

将退出 early suspend 状态

```
echo disk > /sys/power/state
```

命令系统进入 hibernation 状态

```
echo lockname > /sys/power/wake_lock
```

加锁 “lockname”

```
echo lockname > /sys/power/wake_unlock
```

解锁 “lockname”

上述是分别加锁和解锁的命令，一旦系统中所有 wakelock 被解锁，系统就会进入 suspend 状态，可见 Android 中原本使系统 suspend 的操作（echo mem > /sys/power/state 等）被替换成使系统进入 early suspend；而 wake lock 机制成为用户命令系统进入 suspend 状态的唯一途径。