Thinking beyond Source Code

一个喜欢科学的家伙记录生活，技术，爱情，梦想的地方。

- Home
-
- Emacs配置

提交查询内容

Home androidlinux Linux Kernel and Android 休眠与唤醒(中文版)

Linux Kernel and Android 休眠与唤醒(中文版)

四月 18th, 2010 0 Comments/155 hits

Table of Contents

简介

休眠/唤醒在嵌入式Linux中是非常重要的部分,嵌入式设备尽可能的进入休眠状 态来延长电池的续航时间.这篇文章就详细介绍一下Linux中休眠/唤醒是如何工作 的, 还有Android中如何把这部分和Linux的机制联系起来的.

国际化

- English Version: link
- 中文版: link

作者: zhangjiejing <kzjeef#gmail.com> Date: 2010-04-07, http://www.thinksrc.com

版本信息

- Linux Kernel: v2.6.28
- Android: v2.0

对于休眠(suspend)的简单介绍

在Linux中,休眠主要分三个主要的步骤:

1. 冻结用户态进程和内核态任务
2. 调用注册的设备的suspend的回调函数
    - 顺序是按照注册顺序
3. 休眠核心设备和使CPU进入休眠态冻结进程是内核把进程列表中所有的进程的状态都设置为停止,并且保存下所有进程的上下文. 当这些进程被解冻的时候,他们是不知道自己被冻结过的,只是简单的继续执行.如何让Linux进入休眠呢?用户可以通过读写sys文件/sys /power/state 是实现控制系统进入休眠. 比如

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

    命令系统进入休眠. 也可以使用

    ```
    # cat /sys/power/state
    ```

    来得到内核支持哪几种休眠方式.

Linux Suspend 的流程

相关的文件:

你可以通过访问 Linux内核网站 来得到源代码,下面是文件的路径:

- linux_soruce/kernel/power/main.c
- linux_source/kernel/arch/xxx/mach-xxx/pm.c
- linux_source/driver/base/power/main.c

接下来让我们详细的看一下Linux是怎么休眠/唤醒的. Let 's going to see how these happens.

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_state[] 中定义的字符串, 比如"mem", "standby".

然后state_store()会调用enter_state(), 它首先会检查一些状态参数,然后同步文件系统. 下面是代码:

```
/**
 *      enter_state - Do common work of entering low-power state.
 *      @state:         pm_state structure for state we're entering.
 *
 *      Make sure we're the only ones trying to enter a sleep state. Fail
 *      if someone has beat us to it, since we don't want anything weird t
 *      happen when we wake up.
 *      Then, do the setup for suspend, enter the state, and cleaup (after
 *      we've woken up).
 */
static int enter_state(suspend_state_t state)
{
        int error;
        if (!valid_state(state))
                return -ENODEV;
        if (!mutex_trylock(&pm_mutex))
                return -EBUSY;
        printk(KERN_INFO "PM: Syncing filesystems ... ");
        sys_sync();
        printk("done.\n");
        pr_debug("PM: Preparing system for %s sleep\n", pm_states[state]);
        error = suspend_prepare();
        if (error)
                goto Unlock;
        if (suspend_test(TEST_FREEZER))
                goto Finish;
        pr_debug("PM: Entering %s sleep\n", pm_states[state]);
        error = suspend_devices_and_enter(state);
 Finish:
        pr_debug("PM: Finishing wakeup.\n");
        suspend_finish();
 Unlock:
        mutex_unlock(&pm_mutex);
        return error;
}
```

准备, 冻结进程

当进入到suspend_prepare()中以后, 它会给suspend分配一个虚拟终端来输出信 息, 然后广播一个系统要进入suspend的Notify, 关闭掉用户态的helper进程, 然 后一次调用suspend_freeze_processes()冻结所有的进程, 这里会保存所有进程 当前的状态, 也许有一些进程会拒绝进入冻结状态, 当有这样的进程存在的时候, 会导致冻结失败,此函数就会放弃冻结进程,并且解冻刚才冻结的所有进程.

```
/**
 *      suspend_prepare - Do prep work before entering low-power state.
 *
 *      This is common code that is called for each state that we're enter
 *      Run suspend notifiers, allocate a console and stop all processes.
 */
static int suspend_prepare(void)
{
        int error;
        unsigned int free_pages;
        if (!suspend_ops || !suspend_ops->enter)
                return -EPERM;
        pm_prepare_console();
        error = pm_notifier_call_chain(PM_SUSPEND_PREPARE);
        if (error)
                goto Finish;
        error = usermodehelper_disable();
        if (error)
                goto Finish;
        if (suspend_freeze_processes()) {
                error = -EAGAIN;
                goto Thaw;
        }
        free_pages = global_page_state(NR_FREE_PAGES);
        if (free_pages < FREE_PAGE_NUMBER) {
                pr_debug("PM: free some memory\n");
                shrink_all_memory(FREE_PAGE_NUMBER - free_pages);
                if (nr_free_pages() < FREE_PAGE_NUMBER) {
                        error = -ENOMEM;
                        printk(KERN_ERR "PM: No enough memory\n");
                }
        }
        if (!error)
                return 0;
 Thaw:
        suspend_thaw_processes();
        usermodehelper_enable();
 Finish:
        pm_notifier_call_chain(PM_POST_SUSPEND);
        pm_restore_console();
        return error;
}
```

让外设进入休眠

现在, 所有的进程(也包括workqueue/kthread) 都已经停止了, 内核态人物有 可能在停止的时候握有一些信号量, 所以如果这时候在外设里面去解锁这个信号 量有可能会发生死锁, 所以在外设的suspend()函数里面作lock/unlock锁要非常 小心,这里建议设计的时候就不要在suspend()里面等待锁. 而且因为suspend的时候,有一些Log是无法输出的,所以一旦出现问题,非常难调试.

然后kernel在这里会尝试释放一些内存.

最后会调用suspend_devices_and_enter()来把所有的外设休眠, 在这个函数中, 如果平台注册了suspend_pos(通常是在板级定义中定义和注册), 这里就会调用 suspend_ops->begin(), 然后 driver/base/power/main.c 中的 device_suspend()->dpm_suspend() 会被调用,他们会依次调用驱动的suspend() 回调来休眠掉所有的设备.

当所有的设备休眠以后, suspend_ops->prepare()会被调用, 这个函数通常会作 一些准备工作来让板机进入休眠. 接下来Linux,在多核的CPU中的非启动CPU会被关掉, 通过注释看到是避免这些其他的CPU造成race condion,接下来的以后只有一个CPU在运行了.

suspend_ops 是板级的电源管理操作, 通常注册在文件 arch/xxx/mach-xxx/pm.c 中.

接下来, suspend_enter()会被调用, 这个函数会关闭arch irq, 调用 device_power_down(), 它会调用suspend_late()函数, 这个函数是系统真正进入 休眠最后调用的函数, 通常会在这个函数中作最后的检查. 如果检查没问题, 接 下来休眠所有的系统设备和总线, 并且调用 suspend_pos->enter() 来使CPU进入 省电状态. 这时候,就已经休眠了.代码的执行也就停在这里了.

```
/**
 *      suspend_devices_and_enter - suspend devices and enter the desired
 *                                  sleep state.
 *      @state:           state to enter
 */
int suspend_devices_and_enter(suspend_state_t state)
{
        int error, ftrace_save;
        if (!suspend_ops)
                return -ENOSYS;
        if (suspend_ops->begin) {
                error = suspend_ops->begin(state);
                if (error)
                        goto Close;
        }
        suspend_console();
        ftrace_save = __ftrace_enabled_save();
        suspend_test_start();
        error = device_suspend(PMSG_SUSPEND);
        if (error) {
                printk(KERN_ERR "PM: Some devices failed to suspend\n");
                goto Recover_platform;
        }
        suspend_test_finish("suspend devices");
        if (suspend_test(TEST_DEVICES))
                goto Recover_platform;
        if (suspend_ops->prepare) {
                error = suspend_ops->prepare();
                if (error)
                        goto Resume_devices;
        }
        if (suspend_test(TEST_PLATFORM))
                goto Finish;
        error = disable_nonboot_cpus();
        if (!error && !suspend_test(TEST_CPUS))
                suspend_enter(state);
        enable_nonboot_cpus();
 Finish:
        if (suspend_ops->finish)
                suspend_ops->finish();
 Resume_devices:
        suspend_test_start();
        device_resume(PMSG_RESUME);
        suspend_test_finish("resume devices");
        __ftrace_enabled_restore(ftrace_save);
        resume_console();
 Close:
        if (suspend_ops->end)
                suspend_ops->end();
        return error;
 Recover_platform:
        if (suspend_ops->recover)
                suspend_ops->recover();
        goto Resume_devices;
}
```

Resume

如果在休眠中系统被中断或者其他事件唤醒, 接下来的代码就会开始执行, 这个 唤醒的顺序是

和休眠的循序相反的,所以系统设备和总线会首先唤醒,使能系统中 断, 使能休眠时候停止掉的非启动CPU, 以及调用suspend_ops->finish(), 而且 在suspend_devices_and_enter()函数中也会继续唤醒每个设备,使能虚拟终端, 最后调用 suspend_ops->end().

在返回到enter_state()函数中的, 当 suspend_devices_and_enter() 返回以后, 外设已经唤醒了, 但是进程和任务都还是冻结状态, 这里会调用suspend_finish()来解冻这些进程和任务, 而且发出Notify来表示系统已经从suspend状态退出, 唤醒终端.

到这里, 所有的休眠和唤醒就已经完毕了, 系统继续运行了.

Android 休眠(suspend)

在一个打过android补丁的内核中, state_store()函数会走另外一条路,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earlysuspend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

Early suspend 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, 比如重力感应器, 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卡上的文件等. 在嵌入式设备中, 背光是一个很大的电源消耗,所以android会加入这样一种机制.

Late Resume

Late Resume 是和suspend 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, 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就会被调用,并且进入early_suspend状态.

```
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 (debug_mask & DEBUG_USER_STATE) {
                struct timespec ts;
                struct rtc_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.%09lu UTC)\n",
                        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_suspend_work);
        } else if (old_sleep && new_state == PM_SUSPEND_ON) {
                state &= ~SUSPEND_REQUESTED;
                wake_lock(&main_wake_lock);
                queue_work(suspend_work_queue, &late_resume_work);
        }
        requested_suspend_state = new_state;
        spin_unlock_irqrestore(&state_lock, irqflags);
}
```

Early Suspend

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)
{
        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");
        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");
        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);
}
```

Late Resume

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(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");
        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);
}
```

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:

1. WAKE_LOCK_SUSPEND 这种锁会防止系统进入睡眠
2. WAKE_LOCK_IDLE 这种锁不会影响系统的休眠, 作用我不是很清楚.

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

1. 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
2. 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.
3. 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(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 (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:

- 当进入冻结进程的时候, android首先会检查有没有wake lock,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.
- 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输出。

android linux

Related Post

- Linux Kernel and Android Suspend/Resume
- Android Audio System(2): ALSA Layer
- Android Audio System (1)
- android的域名解析

Comments (0) Leave a comment

Name (required)
Mail (will not be published) (required)
Website
=3+8 (required)

Submit Comment (Ctrl+Enter)

Notify me if there is a reply

Config Linux DMA zone memory git: 如何用git-am来合并git format-patch生成的一系列的patch.
想订阅这个博客

Recent Posts

- git: 如何用git-am来合并git format-patch生成的一系列的patch.
- Linux Kernel and Android 休眠与唤醒(中文版)
- Config Linux DMA zone memory
- Linux Kernel and Android Suspend/Resume
- Chromium设置默认的搜索引擎为google.com.tw
- 移动的京东的刷卡体验
- 竖起的显示器
- 在ubuntu中添加debian的源并安装xtables-addons

Recent Comments

-

kzjeef on Android Audio System (1)
anish : "A »

anish on Android Audio System (1)
"AudioFlinger a »

kzjeef on Linux Kernel and Android Suspend/Resume
kasim : Great w »

kasim on Linux Kernel and Android Suspend/Resume
Great work! Thanks f »

kzjeef on Chromium设置默认的搜索引擎为google.com.tw
dd »

Meta

- Login

Tags

GAE GFW android emacs git linux ubuntu 电子商务

Blog roll

- 徐明的博客
- 4G
- 8gdns
- 叶歆昊的个人博客

Top

powered by Micolog Theme by mg12 & eDIKUD .