Power Management Framework in Linux

Pavitrakumar K.M, Vayavya Labs Pvt. Ltd., Bangalore, India.

Pavitra@vayavyalabs.com

Abstract

With the advent of cell phones and tablets that are battery powered devices, Power Management (PM) has become critical. PM means longer battery life, reducing costs and increasing efficiency. Good software framework design complimented with equally good hardware support are key factors for an efficient PM. The PM framework is implemented differently in different OS. In this paper, we present an overview of PM implementation in Linux OS.

Introduction

Power Management (PM) is the practice of saving energy by suspending parts of a computer system when they aren't being used. While a component is "suspended" it is in a nonfunctional low-power state; it might even be turned off completely. A suspended component can be "resumed" (returned to a functional full-power state).

Power management in Linux is performed by the PM core implemented in the OS. The device drivers register the callback functions for suspending and resuming the devices. The decision to either suspend or resume a device lies outside the device driver with the PM core.

1 PM Framework

The Power Management Core in Linux overlooks the power management at the system level. To put the system into sleep state, the PM module communicates with the subsystems which in turn communicate to the devices. The subsystems are responsible for the power management of the devices through the respective device drivers.

As part of the Power Management frameworks the roles & boundaries of the PM core, the subsystems and the drivers are defined. Following are the roles of PM core and the device drivers.

PM Core:
1. Oversees the PM of all the kernel sub-systems
2. Keeps track of the device tree for PM
3. Can communicate with the low level drivers, using driver callbacks
4. Resume all the devices based on wake up events
5. Freeze the user space in system sleep.
6. Maintains pm work queue for devices
7. Exports helper functions which can be used by device drivers.
8. Will take care of the latencies

Device Driver:
1. Implements the framework callbacks viz. suspend and resume.
2. Saves the context of the device before suspension.
3. Drivers can communicate with the PM core using helper functions.
4. Sleeping and Waking up the devices based on the needs of PM core.
5. Handling of the hardware wakeup events if supported.
6. Enabling and disabling of the device interrupts as per PM states.
7. Complete all the I/Os on the device before suspension.

In Linux, there are two types of Power Management frameworks, System sleep and Runtime sleep. As part of the System sleep framework, whole system is suspended or put into low power mode. In Runtime sleep framework only the device under consideration is suspended or put into low power state.

1.1. System Sleep

As part of the System sleep PM framework, the whole system is put to low power mode. This can be done based on user space inputs or device/subsystem activity. Knowing in advance that the whole system is not going to be used in the near future or if the system is idle and no meaningful work is done, results in turning off everything (possibly by force) except for the memory.

System sleep rules
1. Complete system is suspended to RAM
2. User space initiates system sleep
3. User space is frozen as a result of system sleep
4. System is suspended to the lowest power state
5. System suspend can happen at any time
6. Contents of the RAM are preserved during system sleep.

API's that the device driver has to register with PM core to support system sleep are:
`Suspend() and Resume()`

Corresponding functions in the PM core are:
```
int pm_suspend(suspend_state_t state);
void dpm_resume_end(pm_message_t state);
```

In case of system sleep, user space interacts with PM core to suspend the system. For example: Hardware operation of pressing power button or closing laptop lid results in system sleep.

1. **`pm_suspend`** : This function called to suspend the system to RAM. This is exposed to user space as part of an IOCTL and it can be invoked from there.

2. **`enter_suspend`**: This is the function which prepares the system for sleep and also calls the device suspend routines. This is called by "pm_suspend".
3. **`suspend_devices_and_enter`**: This function is called by enter_suspend. It calls the suspend and resume routines of devices.

1.2. Runtime sleep

In case of Runtime Sleep framework, only the device under consideration is turned off. All the other components of the system are in full power and functional state. Runtime (also called Dynamic) Sleep turns off hardware components that are not going to be used in the near future

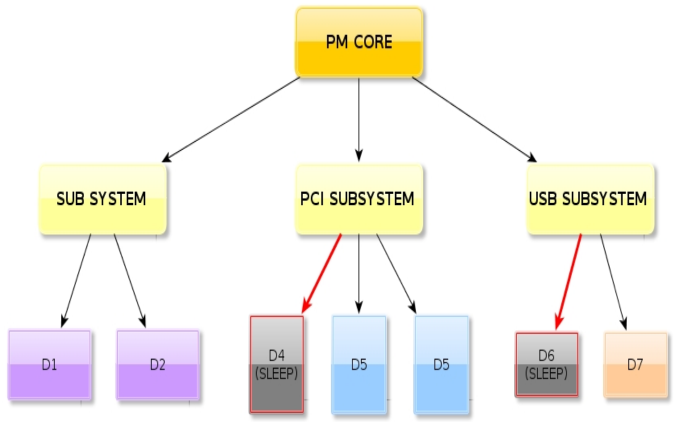

Figure1. Runtime Sleep (devices D4 & D6 in runtime sleep)
.

Runtime sleep rules
1. Only the device in consideration gets suspended.
2. Runtime suspend/resume are transparent to the user space.
3. User space is functional.
4. Runtime suspend is opportunistic.

API's that the device driver had to register with PM core to support runtime sleep are:
```
int (*runtime_suspend)(struct device *dev);
int (*runtime_resume)(struct device *dev);
int (*runtime_idle)(struct device *dev);
```

The PM core helper functions are:
Suspend functions
```
int pm_runtime_suspend(struct device *dev);
int pm_schedule_suspend(struct device *dev,
unsigned int delay);
```
Resume functions
```
int pm_runtime_resume(struct device *dev);
int pm_request_resume(struct device *dev);
```

Notifications of (apparent) idleness

```
int pm_runtime_idle(struct device *dev);
int pm_request_idle(struct device *dev);
```

Taking a reference
```
int pm_runtime_get(struct device
*dev); /* resume request */
int pm_runtime_get_sync(struct device
*dev); /* sync resume */
int pm_runtime_get_noresume(struct device
*dev);
```

Dropping a reference
```
int pm_runtime_put(struct device
*dev); /* idle request */
```

```
int pm_runtime_put_sync(struct device
*dev); /* sync idle */
int pm_runtime_put_noidle(struct device
*dev);
```

In case of runtime sleep, the core provides reference counting facilities. Based on the device usage counter (idle if count=0) the idleness is detected and the individual devices are put in sleep.

The subsystem-level idle callback is executed by the PM core whenever the device appears to be idle, which is indicated to the PM core by two counters, the device's usage counter and the counter of 'active' children of the device. If both of these counters are zero, the PM core executes the subsystem-level idle callback with the device as an argument.

The action performed by a subsystem-level idle callback is totally dependent on the subsystem in question. The expected action is to check if all of the conditions necessary for suspending the device are satisfied. It then queues up a suspend request for the device.

1.3. Suspend/Resume sequence and phases

The sequence of activities for a system suspend and resume are mentioned below.

Suspend Sequence
1. Call notifiers (while user space is still there)
2. Freeze tasks
3. 1st phase of suspending devices
4. Disable device interrupts
5. 2nd phase of suspending devices
6. Disable non-boot CPUs (using CPU hot-plug)
7. Turn interrupts off
8. Execute system core callbacks
9. Put the system to sleep

Table 1 Suspend Sequence

Resume sequence
1. Wakeup signal
2. Run boot CPU’s wakeup code
3. Execute system core callbacks
4. Turn interrupts on
5. Enable non-boot CPUs (using CPU hot-plug)
6. 1st phase of resuming devices
7. Enable device interrupts
8. 2nd phase of suspending devices
9. Thaw tasks
10. Call notifiers (when user space is back)

Table 2 Resume Sequence

The above sequence is broken into multi-phase operations. These phases might be divided among the subsystem and the device drivers. Please refer to table 3 in the appendix A section for details. Most of the phase 2 and some operations in the phase 3 are relevant from the device driver’s point of view. The simplest example is the prepare phase which is mostly for subsystems and not much relevant to the device drivers. A few exceptions may be additional memory allocations for device drivers in prepare phase that have to be done by the driver.

Platform specific operations can be carried out as part of suspend, provided the hardware capabilities are present and supported by the device.

- Gate off clock sources
- Reduce voltage
- Switch off power supplies
- Prepare for wakeup events

Platform specific operations can be carried out as part of resume, provided the hardware capabilities are present and supported by the device.

- Clocks are gated ON
- Switch ON the Clocks and power devices
- Drivers need to re-initialize the device

1.4. PCI Device Power Management

PCI devices can be put into low-power states in two ways. Either by using the device capabilities introduced by the PCI Bus Power Management Interface Specification or with the help of platform firmware, such as an ACPI BIOS.

1.4.1. Native PCI PM

In this approach, the device power state is changed as a result of writing a specific value into one of its standard configuration registers. The PCI PM Spec defines four operating states for devices (D0-D3) and for buses (B0-B3). The higher the number, lesser power is drawn by the device and it longer will be the latency for the device for bus to return to the full-power state.

PCI device drivers have to inform the PCI subsystem of the “device-state” for suspend. Generally suspend routine expects the device to be put into the lowest power state, such as D3 hot. Similarly resume routine puts the device into highest power state, i.e. state D0.

S No	Device states	Power state
1	D0	Full Power State (Mandatory state)
2	D1	Optional state (Low power state compared to D0)
3	D2	Optional state (Low power state compared to D1)
4	D3 Hot	Software accessible state (Low power state compared to D2)
5	D3 Cold	Power off state (not accessible to Software, Reset state since supply voltage (Vcc) is removed)

Table3. PCI Device Power States

1.4.2. Platform based PCI PM

For platform based PM, the ACPI BIOS provides special functions called "control methods" that may be executed by the kernel to perform specific tasks, such as putting a device into a low-power state.

1.5. Remote wakeup

“Remote wakeup” is a mechanism by which suspended devices signal that they should be resumed because of an external event possibly requiring attention.

Some drivers can manage hardware wakeup events, which make the system leave the low-power state. This feature may be enabled or disabled from user space; enabling it may cost some power usage, but let the whole system enter low-power states more often.

Examples of remote wake up events
1. Key press on keyboard
2. Touch on touchscreen
3. Pressing power button
4. Media (SD/MMC) insertion
5. LAN wake up packet
6. USB media insertion in hub

These devices drivers have two flags to control handling of wakeup events.
1. can_wakeup : device_set_wakeup_capable()
2. should_wakeup : device_set_wakeup_enable()

The "can_wakeup" flag records whether the device can physically support wakeup events. The device_set_wakeup_capable() routine affects this flag. The "should_wakeup" flag controls whether the device should try to use its wakeup mechanism. device_set_wakeup_enable() affects this flag. The initial value of should_wakeup is true for the devices where wakeup is enabled.

1.6. ACPI support at driver level

ACPI support will be provided by the operating system. At the device drivers level things remain same with the two power management models, which are system sleep and runtime sleep. No additions are to be made for ACPI support in addition to the suspend/resume routines in the device drivers. ACPI is being dependent on the BIOS data. Since most of the embedded devices do not have BIOS, ACPI is a rarely used and it is not detailed in this document.

2 Conclusion

It is important that the device drivers should be power aware and do everything possible to reduce power utilization. While writing the device drivers the developers have to implement all the power optimization techniques and PM framework.

3 References

[1] Documentation/power/ in the Linux kernel sources.

[2] Rafael J Wysocki https://events.linuxfoundation.org/slides/2011/linuxcon-japan/lcj2011_wysocki2.pdf

[3] Sandeep P. et al: “DDGEN: Device Driver GenerationTool” in IP-ESC’09

Appendix A

Suspend and Resume Sequences

	Suspend Sequence	Resume sequence
Subsystem Suspend / Resume Sequence	1. Call notifiers (while user space is still there) 2. Freeze tasks 3. 1st phase of suspending devices 4. Disable device interrupts 5. 2nd phase of suspending devices 6. Disable non-boot CPUs (using CPU hot-plug) 7. Turn interrupts off 8. Execute system core callbacks 9. Put the system to sleep	1. Wakeup signal 2. Run boot CPU's wakeup code 3. Execute system core callbacks 4. Turn interrupts on 5. Enable non-boot CPUs (using CPU hot-plug) 6. 1st phase of resuming devices 7. Enable device interrupts 8. 2nd phase of suspending devices 9. Thaw tasks 10. Call notifiers (when user space is back)
	Prepare Phase	**Resume_noirq**
Phase 1	1. No new device registration, but devices may be unregistered 2. Memory allocation can be done for proper suspend handling 3. Prepare device for transition 4. Dont put the device in low-power stated	1. Undo the suspend_noirq phase actions 2. Shared interrupts devices : bring device & driver into interrupt enabled state
	Suspend phase	**Resume**
Phase 2	1. Stop IO 2. Save device registers 3. Save enough state to restore/reinit 4. Enable wake up events 5. Put it in low-power mode	1. Undo the actions from suspend phase 2. Get device to ON state where it can do normal IO 3. Get the device to handle DMA & IRQ IO requests 4. New child devices can be registered 5. Drivers need to notice if the device is removed
	suspend_noirq phase	**complete**
Phase 3	1. After IRQ handlers have been disabled 2. Save registers (if anything else was remaining) 3. Suspend devices with shared interrupts (PCI devices) 4. Put device into appropriate low-power mode	Undo actions from prepare phase This phase uses only bus callbacks