power-management.txt

Power Management for USB

Alan Stern <stern@rowland.harvard.edu>

December 11, 2009

What is Power Management?

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) when the kernel needs to use it. (There also are forms of PM in which components are placed in a less functional but still usable state instead of being suspended; an example would be reducing the CPU's clock rate. This document will not discuss those other forms.)

When the parts being suspended include the CPU and most of the rest of the system, we speak of it as a "system suspend". When a particular device is turned off while the system as a whole remains running, we call it a "dynamic suspend" (also known as a "runtime suspend" or "selective suspend"). This document concentrates mostly on how dynamic PM is implemented in the USB subsystem, although system PM is covered to some extent (see Documentation/power/*.txt for more information about system PM).

Note: Dynamic PM support for USB is present only if the kernel was built with CONFIG_USB_SUSPEND enabled (which depends on CONFIG_PM_RUNTIME). System PM support is present only if the kernel was built with CONFIG_SUSPEND or CONFIG_HIBERNATION enabled.

What is Remote Wakeup?

When a device has been suspended, it generally doesn't resume until the computer tells it to. Likewise, if the entire computer has been suspended, it generally doesn't resume until the user tells it to, say by pressing a power button or opening the cover.

However some devices have the capability of resuming by themselves, or asking the kernel to resume them, or even telling the entire computer to resume. This capability goes by several names such as "Wake On LAN"; we will refer to it generically as "remote wakeup". When a device is enabled for remote wakeup and it is suspended, it may resume itself (or send a request to be resumed) in response to some external event. Examples include a suspended keyboard resuming when a key is pressed, or a suspended USB hub resuming when a device is plugged in.

When is a USB device idle?

A device is idle whenever the kernel thinks it's not busy doing anything important and thus is a candidate for being suspended. The exact definition depends on the device's driver; drivers are allowed to declare that a device isn't idle even when there's no actual communication taking place. (For example, a hub isn't considered idle unless all the devices plugged into that hub are already suspended.) In addition, a device isn't considered idle so long as a program keeps its usbfs file open, whether or not any I/O is going on.

If a USB device has no driver, its usbfs file isn't open, and it isn't being accessed through sysfs, then it definitely is idle.

Forms of dynamic PM

Dynamic suspends occur when the kernel decides to suspend an idle device. This is called "autosuspend" for short. In general, a device won't be autosuspended unless it has been idle for some minimum period of time, the so-called idle-delay time.

Of course, nothing the kernel does on its own initiative should prevent the computer or its devices from working properly. If a device has been autosuspended and a program tries to use it, the kernel will automatically resume the device (autoresume). For the same reason, an autosuspended device will usually have remote wakeup enabled, if the device supports remote wakeup.

It is worth mentioning that many USB drivers don't support autosuspend. In fact, at the time of this writing (Linux 2.6.23) the only drivers which do support it are the hub driver, kaweth, asix, usblp, usblcd, and usb-skeleton (which doesn't count). If a non-supporting driver is bound to a device, the device won't be autosuspended. In effect, the kernel pretends the device is never idle.

We can categorize power management events in two broad classes: external and internal. External events are those triggered by some agent outside the USB stack: system suspend/resume (triggered by userspace), manual dynamic resume (also triggered by userspace), and remote wakeup (triggered by the device). Internal events are those triggered within the USB stack: autosuspend and autoresume. Note that all dynamic suspend events are internal; external agents are not allowed to issue dynamic suspends.

The user interface for dynamic PM

The user interface for controlling dynamic PM is located in the power/ subdirectory of each USB device's sysfs directory, that is, in /sys/bus/usb/devices/.../power/ where "..." is the device's ID. The relevant attribute files are: wakeup, control, and autosuspend. (There may also be a file named "level"; this file was deprecated as of the 2.6.35 kernel and replaced by the "control" file.)

power/wakeup

This file is empty if the device does not support remote wakeup. Otherwise the file contains either the word "enabled" or the word "disabled", and you can write those words to the file. The setting determines whether or not remote wakeup will be enabled when the device is next suspended. (If the setting is changed while the device is suspended, the change won't take effect until the following suspend.)

power/control

This file contains one of two words: "on" or "auto". You can write those words to the file to change the device's setting.

"on" means that the device should be resumed and autosuspend is not allowed. (Of course, system suspends are still allowed.)

"auto" is the normal state in which the kernel is allowed to autosuspend and autoresume the device.

(In kernels up to 2.6.32, you could also specify "suspend", meaning that the device should remain suspended and autoresume was not allowed. This setting is no longer supported.)

power/autosuspend

This file contains an integer value, which is the number of seconds the device should remain idle before the kernel will autosuspend it (the idle-delay time). The default is 2. 0 means to autosuspend as soon as the device becomes idle, and negative values mean never to autosuspend. You can write a number to the file to change the autosuspend idle-delay time.

Writing "-1" to power/autosuspend and writing "on" to power/control do essentially the same thing -- they both prevent the device from being autosuspended. Yes, this is a redundancy in the API.

(In 2.6.21 writing "0" to power/autosuspend would prevent the device from being autosuspended; the behavior was changed in 2.6.22. The power/autosuspend attribute did not exist prior to 2.6.21, and the power/level attribute did not exist prior to 2.6.22. power/control was added in 2.6.34.)

Changing the default idle-delay time

The default autosuspend idle-delay time is controlled by a module parameter in usbcore. You can specify the value when usbcore is

loaded. For example, to set it to 5 seconds instead of 2 you would do:

```
modprobe usbcore autosuspend=5
```

Equivalently, you could add to /etc/modprobe.conf a line saying:

```
options usbcore autosuspend=5
```

Some distributions load the usbcore module very early during the boot process, by means of a program or script running from an initramfs image. To alter the parameter value you would have to rebuild that image.

If usbcore is compiled into the kernel rather than built as a loadable module, you can add

```
usbcore.autosuspend=5
```

to the kernel's boot command line.

Finally, the parameter value can be changed while the system is running. If you do:

```
echo 5 >/sys/module/usbcore/parameters/autosuspend
```

then each new USB device will have its autosuspend idle-delay initialized to 5. (The idle-delay values for already existing devices will not be affected.)

Setting the initial default idle-delay to -1 will prevent any autosuspend of any USB device. This is a simple alternative to disabling CONFIG_USB_SUSPEND and rebuilding the kernel, and it has the added benefit of allowing you to enable autosuspend for selected devices.

Warnings

The USB specification states that all USB devices must support power management. Nevertheless, the sad fact is that many devices do not support it very well. You can suspend them all right, but when you try to resume them they disconnect themselves from the USB bus or they stop working entirely. This seems to be especially prevalent among printers and scanners, but plenty of other types of device have the same deficiency.

For this reason, by default the kernel disables autosuspend (the power/control attribute is initialized to "on") for all devices other than hubs. Hubs, at least, appear to be reasonably well-behaved in this regard.

(In 2.6.21 and 2.6.22 this wasn't the case. Autosuspend was enabled by default for almost all USB devices. A number of people experienced problems as a result.)

This means that non-hub devices won't be autosuspended unless the user or a program explicitly enables it. As of this writing there aren't any widespread programs which will do this; we hope that in the near future device managers such as HAL will take on this added responsibility. In the meantime you can always carry out the necessary operations by hand or add them to a udev script. You can also change the idle-delay time; 2 seconds is not the best choice for every device.

If a driver knows that its device has proper suspend/resume support, it can enable autosuspend all by itself. For example, the video driver for a laptop's webcam might do this, since these devices are rarely used and so should normally be autosuspended.

Sometimes it turns out that even when a device does work okay with autosuspend there are still problems. For example, there are experimental patches adding autosuspend support to the usbhid driver, which manages keyboards and mice, among other things. Tests with a number of keyboards showed that typing on a suspended keyboard, while causing the keyboard to do a remote wakeup all right, would nonetheless frequently result in lost keystrokes. Tests with mice showed that some of them would issue a remote-wakeup request in response to button presses but not to motion, and some in response to neither.

The kernel will not prevent you from enabling autosuspend on devices that can't handle it. It is even possible in theory to damage a device by suspending it at the wrong time -- for example, suspending a USB hard disk might cause it to spin down without parking the heads. (Highly unlikely, but possible.) Take care.

The driver interface for Power Management

The requirements for a USB driver to support external power management are pretty modest; the driver need only define

```
.suspend
.resume
.reset_resume
```

methods in its usb_driver structure, and the reset_resume method is optional. The methods' jobs are quite simple:

> The suspend method is called to warn the driver that the device is going to be suspended. If the driver returns a negative error code, the suspend will be aborted. Normally the driver will return 0, in which case it must cancel all outstanding URBs (usb_kill_urb()) and not submit any more.
>
> The resume method is called to tell the driver that the device has been resumed and the driver can return to normal operation. URBs may once more be submitted.

The reset_resume method is called to tell the driver that
the device has been resumed and it also has been reset.
The driver should redo any necessary device initialization,
since the device has probably lost most or all of its state
(although the interfaces will be in the same altsettings as
before the suspend).

If the device is disconnected or powered down while it is suspended, the disconnect method will be called instead of the resume or reset_resume method. This is also quite likely to happen when waking up from hibernation, as many systems do not maintain suspend current to the USB host controllers during hibernation. (It's possible to work around the hibernation-forces-disconnect problem by using the USB Persist facility.)

The reset_resume method is used by the USB Persist facility (see Documentation/usb/persist.txt) and it can also be used under certain circumstances when CONFIG_USB_PERSIST is not enabled. Currently, if a device is reset during a resume and the driver does not have a reset_resume method, the driver won't receive any notification about the resume. Later kernels will call the driver's disconnect method; 2.6.23 doesn't do this.

USB drivers are bound to interfaces, so their suspend and resume methods get called when the interfaces are suspended or resumed. In principle one might want to suspend some interfaces on a device (i.e., force the drivers for those interface to stop all activity) without suspending the other interfaces. The USB core doesn't allow this; all interfaces are suspended when the device itself is suspended and all interfaces are resumed when the device is resumed. It isn't possible to suspend or resume some but not all of a device's interfaces. The closest you can come is to unbind the interfaces' drivers.

The driver interface for autosuspend and autoresume

To support autosuspend and autoresume, a driver should implement all three of the methods listed above. In addition, a driver indicates that it supports autosuspend by setting the .supports_autosuspend flag in its usb_driver structure. It is then responsible for informing the USB core whenever one of its interfaces becomes busy or idle. The driver does so by calling these six functions:

```
int  usb_autopm_get_interface(struct usb_interface *intf);
void usb_autopm_put_interface(struct usb_interface *intf);
int  usb_autopm_get_interface_async(struct usb_interface *intf);
void usb_autopm_put_interface_async(struct usb_interface *intf);
void usb_autopm_get_interface_no_resume(struct usb_interface *intf);
void usb_autopm_put_interface_no_suspend(struct usb_interface *intf);
```

The functions work by maintaining a usage counter in the usb_interface's embedded device structure. When the counter is > 0 then the interface is deemed to be busy, and the kernel will not autosuspend the interface's device. When the usage counter is = 0 then the interface is considered to be idle, and the kernel may

autosuspend the device.

(There is a similar usage counter field in struct usb_device,
associated with the device itself rather than any of its interfaces.
This counter is used only by the USB core.)

Drivers need not be concerned about balancing changes to the usage
counter; the USB core will undo any remaining "get"s when a driver
is unbound from its interface. As a corollary, drivers must not call
any of the usb_autopm_* functions after their diconnect() routine has
returned.

Drivers using the async routines are responsible for their own
synchronization and mutual exclusion.

    usb_autopm_get_interface() increments the usage counter and
    does an autoresume if the device is suspended. If the
    autoresume fails, the counter is decremented back.

    usb_autopm_put_interface() decrements the usage counter and
    attempts an autosuspend if the new value is = 0.

    usb_autopm_get_interface_async() and
    usb_autopm_put_interface_async() do almost the same things as
    their non-async counterparts. The big difference is that they
    use a workqueue to do the resume or suspend part of their
    jobs. As a result they can be called in an atomic context,
    such as an URB's completion handler, but when they return the
    device will generally not yet be in the desired state.

    usb_autopm_get_interface_no_resume() and
    usb_autopm_put_interface_no_suspend() merely increment or
    decrement the usage counter; they do not attempt to carry out
    an autoresume or an autosuspend. Hence they can be called in
    an atomic context.

The simplest usage pattern is that a driver calls
usb_autopm_get_interface() in its open routine and
usb_autopm_put_interface() in its close or release routine. But other
patterns are possible.

The autosuspend attempts mentioned above will often fail for one
reason or another. For example, the power/control attribute might be
set to "on", or another interface in the same device might not be
idle. This is perfectly normal. If the reason for failure was that
the device hasn't been idle for long enough, a timer is scheduled to
carry out the operation automatically when the autosuspend idle-delay
has expired.

Autoresume attempts also can fail, although failure would mean that
the device is no longer present or operating properly. Unlike
autosuspend, there's no idle-delay for an autoresume.

    Other parts of the driver interface

Drivers can enable autosuspend for their devices by calling

        usb_enable_autosuspend(struct usb_device *udev);

in their probe() routine, if they know that the device is capable of suspending and resuming correctly. This is exactly equivalent to writing "auto" to the device's power/control attribute. Likewise, drivers can disable autosuspend by calling

        usb_disable_autosuspend(struct usb_device *udev);

This is exactly the same as writing "on" to the power/control attribute.

Sometimes a driver needs to make sure that remote wakeup is enabled during autosuspend. For example, there's not much point autosuspending a keyboard if the user can't cause the keyboard to do a remote wakeup by typing on it. If the driver sets intf->needs_remote_wakeup to 1, the kernel won't autosuspend the device if remote wakeup isn't available or has been disabled through the power/wakeup attribute. (If the device is already autosuspended, though, setting this flag won't cause the kernel to autoresume it. Normally a driver would set this flag in its probe method, at which time the device is guaranteed not to be autosuspended.)

If a driver does its I/O asynchronously in interrupt context, it should call usb_autopm_get_interface_async() before starting output and usb_autopm_put_interface_async() when the output queue drains. When it receives an input event, it should call

        usb_mark_last_busy(struct usb_device *udev);

in the event handler. This sets udev->last_busy to the current time. udev->last_busy is the field used for idle-delay calculations; updating it will cause any pending autosuspend to be moved back. Most of the usb_autopm_* routines will also set the last_busy field to the current time.

Asynchronous operation is always subject to races. For example, a driver may call one of the usb_autopm_*_interface_async() routines at a time when the core has just finished deciding the device has been idle for long enough but not yet gotten around to calling the driver's suspend method. The suspend method must be responsible for synchronizing with the output request routine and the URB completion handler; it should cause autosuspends to fail with -EBUSY if the driver needs to use the device.

External suspend calls should never be allowed to fail in this way, only autosuspend calls. The driver can tell them apart by checking the PM_EVENT_AUTO bit in the message.event argument to the suspend method; this bit will be set for internal PM events (autosuspend) and clear for external PM events.

        Mutual exclusion

For external events -- but not necessarily for autosuspend or autoresume -- the device semaphore (udev->dev.sem) will be held when a suspend or resume method is called. This implies that external suspend/resume events are mutually exclusive with calls to probe, disconnect, pre_reset, and post_reset; the USB core guarantees that this is true of autosuspend/autoresume events as well.

If a driver wants to block all suspend/resume calls during some critical section, the best way is to lock the device and call usb_autopm_get_interface() (and do the reverse at the end of the critical section). Holding the device semaphore will block all external PM calls, and the usb_autopm_get_interface() will prevent any internal PM calls, even if it fails. (Exercise: Why?)

Interaction between dynamic PM and system PM

Dynamic power management and system power management can interact in a couple of ways.

Firstly, a device may already be autosuspended when a system suspend occurs. Since system suspends are supposed to be as transparent as possible, the device should remain suspended following the system resume. But this theory may not work out well in practice; over time the kernel's behavior in this regard has changed.

Secondly, a dynamic power-management event may occur as a system suspend is underway. The window for this is short, since system suspends don't take long (a few seconds usually), but it can happen. For example, a suspended device may send a remote-wakeup signal while the system is suspending. The remote wakeup may succeed, which would cause the system suspend to abort. If the remote wakeup doesn't succeed, it may still remain active and thus cause the system to resume as soon as the system suspend is complete. Or the remote wakeup may fail and get lost. Which outcome occurs depends on timing and on the hardware and firmware design.