OpenBSD's New Suspend and Resume Framework

Paul Irofti

March 11, 2011

Abstract

Suspend and resume support in OpenBSD was almost complete in the 4.8 Release. During the development, a lot had to be changed - it was a long process, starting from *acpi*(4) and *apm*(4) changes, down into the low level parts of *autoconf*(9) and upwards into the device drivers. What started as i386 and amd64 targeted development turned into a machine independent framework that can now be used by other architectures. Currently, loongson is one such architecture that is still a work in progress.

1 Device Configuration

The devices found on a machine running OpenBSD are handled by the kernel via the *autoconf*(9) mechanism. Each new device is added, based on hierarchy, into a tree structure.

If, for example, a memory stick is plugged into one of the machine's USB ports, it would likely be visible to the user as an *sd*(4) disk. The *sd*(4) driver would assign a new device, such as *sd*0, to the memory stick and attach itself to a SCSI bus. The SCSI bus in this case is provided by the USB mass storage support driver, *umass*(4). Presence of *umass*(4) devices implies presence of at least one USB hub (implemented by *uhub*(4)). Following this chain further, *uhub*(4) devices attach to *usb*(4) controllers. *usb*(4) controllers are typically connected to *pci*(4) buses, which ultimately connect back to *mainbus*(4). In total, the connection path looks like *sd*(4) → *scsibus*(4) → *umass*(4) → *uhub*(4) → *usb*(4) → *ehci*(4) → *pci*(4) → *mainbus*(4) → *root*.

Each device has a set of specific functions that configure the device. Every time a new device is found its first matched against the existing drivers and then, when identified, the device is attached to the device tree (as illustrated previously with the *usb*(4) example). After the device is activated it becomes ready for use.

Upon shutdown or when unplugged, a device is first deactivated (notifying parents and dependent sub-systems) and then detached from the system.

All of these actions are implemented by functions inside the *autoconf*(9) system and are stored by drivers in a cfattach structure. For example the *run*(4) driver has *run_match*() that performs a *usb_lookup*() for vendor-product tags, *run_attach*() sets

up the device capabilities and fetches the required interfaces, *run_activate*() handles the activate and deactivate cases and finally *run_detach*() marks the interface down and frees the RX and TX rings.

When a machine is suspended, most of its hardware gets powered off and the critical parts are moved into a lower power state. This allows them to do the minimum amount of tasks necessary for saving the system state and signal the machine to wake-up when requested.

Thus the best place for drivers to process the suspend and resume signals would be inside their activate functions, similar to the activate/deactivate logic.

2 Activate functions

In order to be able to process suspend and resume signals from inside the driver's activate function the *autoconf*(9) architecture needs to be expanded to act upon such events by broadcasting them down the device tree.

The config family functions (attach, detach, activate and deactivate) now have a new member called $config\_suspend()$ that will signal the driver with a suspend or resume action, depending on what the caller requested.

The $config\_activate\_children()$ function also needs to be enhanced by expanding it for the new use cases. The function will call $config\_suspend(dev, action)$ while it iterates through the device's children. If one refuses to suspend, the function will try to resume the previous devices in an attempt to restore the system state and signal the error.

After that it becomes the work of the drivers to do the proper thing upon suspend and resume. Some hardware is more robust than others, in that the documentation isn't flawed and does what the power management section says it does; others can't be trusted and the kernel driver needs to save as much state in software as needed to restore it upon resume.

Extended tests on multiple configurations and machine types ensured that the drivers for poorly documented hardware became more robust and stable by discovering new registers that needed to be saved and uncovering a proper flow in which the suspend and resume actions needed to take place.

Quiesce Some devices need to be notified beforehand about suspending in order to finish or stop currently running jobs. The context in which these tasks are handled is very important. To do this, some drivers require enabled interrupts, a running clock and the ability to be provided longer hardware response delays (to handle possibly long-running tasks such as disk I/O).

For such devices, the drivers can receive another action in their activate function - QUIESCE. The term is mainly used in literature concerning storage and databases; it describes the action of pausing or modifying a given process so that data consistency can be achieved.

Tasks might consist of finishing the current disk I/O, dumping audio buffers, video handling for VT switching out of X, waiting for other dependent operations to succeed and any other actions the driver should undergo while the machine is in a 'normal' running state.

3 APM and ACPI

With a driver framework and a way to properly signal the devices, from a hierarchy and action point of view, the machine independent kernel implementation is complete. The rest is up to the machine's power management capabilities. Currently there are two ways of putting a machine in a low power state: APM and ACPI.

3.1 APM

Most architectures give almost complete responsibility to the hardware or the BIOS [3] to handle the machine specific tasks for suspend and resume. This results in very little work on the operating system side of things.

In this case a userland daemon polls events from the user and hands them to the kernel for processing. With the old suspend framework this included handling some specific tasks that needed to be done by devices found on the given architecture. This was simplified and moved in the machine independent part of the kernel.

The code flow for most of the APM architectures is now very similar and easy to maintain because of that. It usually boils down to the following actions:

On suspend

- *wsdisplay_suspend*() — suspend the console display
- *bufq_quiesce*() — prevent any new I/O from reaching disk devices
- $config\_suspend(QUIESCE)$ — start broadcasting the quiesce message to the device tree so that their activate functions can stop the current operations
- *splhigh*() — set the system priority level to high, blocking all hard and soft interrupts
- *disableintr*() — low level system specific instructions to disable all interrupts
- $config\_suspend(SUSPEND)$ — now it is safe to call the specific suspend code for the drivers
- $sys\_platform \rightarrow suspend()$ — almost done, only the low level platform specific parts need to be executed before the system is fully suspended; this is also the place where wake-up triggers are set, such as power buttons, network cards, special keyboard events, etc.

On resume

- $sys\_platform \rightarrow resume()$ — when a wake-up event is received the system awakes by first doing the low level platform specific parts (e.g. resetting the proper power plane and powering the CPU and the fans)
- $config\_suspend(RESUME)$ — while the interrupts are still disabled signal the drivers to restore state and reinitialize the needed registers
- *enableintr*() — call the low level instructions set to permit all interrupts
- *splx*() — restore the system priority level prior to going to splhigh() on suspend

- *bufq_restart*() — restart the buffer queue mechanism, allowing I/O
- *wsdisplay_resume*() — turn on the console display

3.2 ACPI

OpenBSD has its own ACPI [1] implementation. The only other open source alternative which most operating systems are using is Intel's ACPICA [2]. There is also a third closed-source implementation used in Microsoft Windows.

ACPI treats things differently than APM. It acts like a bridge between the BIOS and the operating system allowing a lot of flexibility and giving almost full control to the latter.

This makes the implementation more complex and thus increases the machine dependent part of the code. The kernel must take care of everything, from setting up its own resume trampoline (containing the real-mode → protected-mode switch), to making sure the suspend lamp is blinking.

All of this relies in the end on specific AML methods that should be implemented according to the ACPI specification. AML is the language in which the specification is implemented for a particular machine. The code differs a lot between models even from the same manufacturer, so the kernel can only make use of the methods from the specification and create special product-specific drivers for quirks and documentation violations (which are encountered frequently).

The ACPI sub-system follows the same logic flow as APM. It creates a fake APM notification ioctl, *acpiioctl*(), that handles the same commands as a regular APM machine. In case the user requests the machine to be suspended, an ACPI task is added that calls *acpi_sleep_task*() which puts the system in S3 (suspend to RAM).

The transition from S0 (normal operation state) to S3 looks something like this:

- *acpi_sleep_task*(*S*3) — checks if there's been a state transition requested and if not proceeds to update runtime information like battery life
- *acpi_sleep_mode*(*S*3) — handles state changes, in case of suspend it sets up the sleep state and then calls the machine dependent parts to finish the switch to S3
- *acpi_prepare_sleep_state*(*S*3) — this is almost identical to the APM suspend flow described earlier with a few ACPI specific tasks in-between:
  - TTS method — AML method that transitions the machine to a given state and should be called before notifying the devices about suspending
  - PTS method — AML method that prepares the machine to sleep and should be called after the devices were notified about the suspend transition.
  - SST — system status indicator used to reset the lights to show that the platform is in a suspended state (e.g. blinking moon or LED, or other visual indicator)

  - GTS method — AML method that permits the ACPI system firmware to perform any required system specific functions prior to entering S3
  - PM registers — set the power management registers accordingly
  - GPE wake registers — make sure all wake signals are enabled

- *acpi_sleep_machdep*($S3$) — go to the machine dependent code and execute the required suspend tasks

- *acpi_enter_sleep_state*($S3$) — in the end do the last bit of fiddling with ACPI specific power management registers and finally suspend the whole system

On resume the system starts from an ACPI trampoline that runs in real-mode. Here the video is reenabled, then the system goes to protected mode and enables paging, afterwards the saved CPU registers are restored and a jump is taken to the point where the system left off in the ACPI code during suspend.

Besides the actions described in the APM case, the resume path takes care of the following ACPI specific tasks:

- clearing of resetting the ACPI PM registers

- calling specific AML methods to transition back to S0

- reseting the system status indicators

- enabling the runtime GPEs (events that are used to indicate something of interest to the ACPI subsystem took place, like a lid closing, a power button press, etc)

4 Issues

There are still some issues in the current framework that are due to poor documentation or the lack of it altogether.

The ACPI specification is not followed by most vendors. It seems it is just a point of reference at best. Some machines call magic CMOS methods to do stuff and create stubs for the related specification methods. Worse yet, there are different AML methods pending on ACPI heuristics done by the system to determine which operating system is running. This is why OpenBSD has to register as Windows to the BIOS in order to get the proper methods since many implementations will provide empty or broken methods for other operating systems.

Hardware documentation is an old topic but also relevant in this case. Most of the development for some devices has been a "hide and seek" process with the device registers in an effort to find out which ones to save, which bits to set and in what order.

Another challenging issue was video reposting (reinitialization on resume). Some video cards need to be reenabled from real-mode on resume. Others require the entire BIOS video code to be executed, thus a real-mode emulator was added to the kernel and the BIOS code is now mapped and ran through it at resume time.

The best case scenario is represented by some video cards that don't need any work outside their regular driver activate functions. And then there's nVIDIA cards that

just don't work and have no documentation.

To determine which video repost path to use, the kernel uses a heuristic to determine, based on PCI ID, which video card is present and how to act on resume.

5 Conclusions

The new framework is flexible and robust. New architectures need only add their machine dependent code and respect the call flow of the machine independent bits. For new devices the code is also encapsulated in their activate functions and the rest is taken care of by the *autoconf*(9) framework.

The ACPI implementation is a bit more sensitive, in the sense that newer machines will probably bring in new quirks and violations of the standard. On the other hand the machine independent part is pretty stable at this point and is probably less affected by these drawbacks.

There is still work to be done in this area. Hibernate (S4) support is almost ready for prime-time. Nouveau [4] have tackled the NVIDIA video problems, describing a horrible hardware implementation that at least seems to be fixable in software. New platform ports like Loongson exhibit more suspend and resume bugs in device drivers like glxpcib(4) and bring out new challenges. Likewise, Itanium will be yet another interesting project as it will be another ACPI based platform.

References

[1] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation. *Advanced Configuration and Power Interface Specification, Revision 4.0*. 2009.

[2] Intel Corporation. *ACPI Component Architecture User Guide and Programmer Reference*. Intel Corporation, 2010.

[3] Intel Corporation and Microsoft Corporation. *Advanced Power Management (APM) BIOS Interface Specification*. 1996.

[4] FreeDesktop Nouveau. Accelerated opensource driver for nvidia cards. `http://nouveau.freedesktop.org/`. [Online; accessed 12-February-2011].