Linux Kernel Power Management

Patrick Mochel
Open Source Development Labs
mochel@osdl.org

Abstract

Power management is the process by which the overall consumption of power by a computer is limited based on user requirements and policy. Power management has become a hot topic in the computer world in recent years, as laptops have become more commonplace and users have become more conscious of the environmental and financial effects of limited power resources.

While there is no such thing as perfect power management, since all computers must use some amount of power to run, there have been many advances in system and software architectures to conserve the amount of power being used. Exploiting these features is key to providing good system- and device-level power management.

This paper discusses recent advances in the power management infrastructure of the Linux kernel that will allow Linux to fully exploit the power management capabilities of the various platforms that it runs on. These advances will allow the kernel to provide equally great power management, using a simple interface, regardless of the underlying archtitecture.

This paper covers the two broad areas of power management—System Power Management (SPM) and Device Power Management (DPM). It describes the major concepts behind both subjects and describes the new kernel infrastructure for implementing both. It also discusses the mechanism for implementing hibernation, otherwise known as suspend-to-disk, support for Linux.

1 Overview

Benefits of Power Management

A sane power management infrastructure provides many benefits to the kernel, and not only in the obvious areas.

Battery-powered devices, such as embedded devices, handhelds, and laptops reap most of the rewards of power management, since the more conservative the draw on the battery is, the longer it will last.

System power management decreases boot time of a system, by restoring previously saved state instead of reinitializing the entire system. This conserves battery life on mobile devices by reducing the annoying wait for the computer to boot into a useable state.

Recently, power management concepts have begun to filter into less obvious places, like the enterprise. In a rack of servers, some servers may power down during idle times, and power back up when needed again to fulfill network requests. While the power consumption of a single server is but a drop in the water, being able to conserve the power draw of dozens or hundreds of computers could save a company a significant amount of money.

Also, at the lower-level, power management may be used to provide emergency reaction to a critical system state, such as crossing a predefined thermal threshold or reaching a critically low battery state. The same concept can be applied when triggering a critical software state, like an Oops or a BUG() in the kernel.

System and Device Power Management

There are two types of power management that the OS must handle—System Power Management and Device Power Management.

Device Power Management deals with the process of placing individual devices into low-power states while the system is running. This allows a user to conserve power on devices that are not currently being used, such as the sound device in my laptop while I write this paper.

Individual device power management may be invoked explicitly on devices, or may happen automatically after a device has been idle for a set of amount of time. Not all devices support run-time power management, but those that do must export some mechanism for controlling it in order to execute the user's policy decisions.

System Power Management is the process by which the entire system is placed into a low-power state. There are several power states that a system may enter, depending on the platform it is running on. Many are similar across platforms, and will be discussed in detail later. The general concept is that the state of the running system is saved before the system is powered down, and restored once the system has regained power. This prevents the system from performing an entire shutdown and startup sequence.

System power management may be invoked for a number of reasons. It may automatically enter a low-power state after it has been idle for some amount of time, after a user closes a lid on a laptop, or when some critical state has been reached. These are also policy decisions that are up to the user to configure and require some global mechanism for controlling.

2 Device Power Management

Overview

Device power management in the kernel is made possible by the new driver model in the 2.5 kernel. In fact, the driver model was inspired by the requirement to implement decent power management in the kernel. The new driver model allows generic kernel to communicate with every device in the system, regardless of the bus the device resides on, or the class it belongs to.

The driver model also provides a hierarchical representation of the devices in the system. This is key to power management, since the kernel cannot power down a device that another device, that isn't powered down, relies on for power. For example, the system cannot power down a parent device whose children are still powered up and depend on their parent for power.

In its simplest form, device power management consists of a description of the state a device is in, and a mechanism for controlling those states. Device power states are described as 'D' states, and consist of states D0-D3, inclusive. This device state representation is inspired by the PCI device specification and the ACPI specification [ACPI]. Though not all device types define power states in this way, this representation can map on to all known device types.

Each D state represents a tradeoff between the amount of power a device is consuming and how functional a device is. In a lower power state (represented by a higher digit following

D), some amount of power to a device is lost. This means that some of the device's operating state is lost, and must be restored by its driver when returning to the D0 state.

D0 represents the state when the device is fully powered on and ready for, or in, use. This state is implicitly supported by every device, since every device may be powered on at some point while the system is running. In this state, all units of a device are powered on, and no device state is lost.

D3 represents the state when the device is off. This state is also implicitly supported by every device, since every device is implicitly powered off when the system is powered off. In this state, all device context is lost and must be restored before using the device again. This usually means the device must also be completely reinitialized.

The PCI Power Management spec goes on to define D3hot as a D3 state that is entered via driver control and D3cold that is entered when the entire system is powered down. In D3hot, the device may not lose all operating power, requiring less restoration that must take place. This is however, device-dependent. The kernel does not distinguish between the two, though a driver theoretically could take extra steps to do so.

D1 and D2 are intermediate power states that are optionally supported by a device. In each case, the device is not functional, but not entirely powered off. In order to bring the device back to an operating state, less work is required than reviving the device from D3. In D1, more power is consumed than in D2, but more device context is preserved.

A device's power management information is stored in `struct device_pm`:

```
struct device_pm {
#ifdef CONFIG_PM
    dev_power_t     power_state;
    u8              * saved_state;
    atomic_t        depend;
    atomic_t        disable;
    struct kobject  kobj;
#endif
};
```

`struct device` contains a statically allocated `device_pm` object. The PM configuration dependency guarantees the overhead for the structure is nil when power management support is not compiled in.

The kernel defines the following power states in `include/linux/pm.h`:

```
typedef enum {
    DEVICE_PM_ON,
    DEVICE_PM_INT1,
    DEVICE_PM_INT2,
    DEVICE_PM_OFF,
    DEVICE_PM_UNKNOWN,
} dev_power_t;
```

When a device is registered, its initial power state is set to `DEVICE_PM_UNKNOWN`. The device driver may query the device and initialize the known power state using

```
void device_pm_init_power_state(
  struct device * dev,
  dev_power_t state);
```

Controlling a Device's State

A device's power state may be controlled by the `suspend()` and `resume()` methods in `struct device_driver`:

```
int (*suspend)(struct device * dev,
               u32 state, u32 level);
int (*resume) (struct device * dev,
               u32 level);
```

These methods may be initialized by the low-level device driver, though they are typically initialized at registration time by the bus driver that the driver belongs to. The bus's functions should forward power management requests to the bus-specific driver, modifying the semantics where necessary.

This model is used to provide the easiest route when converting to the new driver model. However, a device driver's explicit initialization of these methods will be honored.

The same methods are called during individual device power management transitions and system power management transitions.

There are two steps to suspend a device and two steps to resume it. In order to suspend a device, two separate calls are made to the suspend() method—one to save state, and another to power the device down. Conversely, one call is made to the resume() method to power the device up, and another to restore device state.

These steps are encoded:

```
enum {
    SUSPEND_SAVE_STATE,
    SUSPEND_POWER_DOWN,
};

enum {
    RESUME_POWER_ON,
    RESUME_RESTORE_STATE,
};
```

and are passed as the 'level' parameter to each method.

During the `SUSPEND_SAVE_STATE` call, the driver is expected to stop all device requests and save all relevant device context based on the state the device is entering.

This call is made in process context, so the driver may sleep and allocate memory to save state. However during system suspend, backing swap devices may have already been powered down, so drivers should use `GFP_ATOMIC` when allocating memory.

`SUSPEND_POWER_DOWN` is used only to physically power the device down. This call has some caveats, and drivers must be aware of them. Interrupts will be disabled when this routine is called. However, during run-time device power management, interrupts will be reenabled once the call returns. Some devices are known to cause problems once they are powered down and interrupts reenabled—e.g. flooding the system with interrupts. Drivers should be careful not to service power management requests for devices known to be buggy.

During system power management, interrupts are disabled and remain disabled while powering down all devices in the system.

The resume sequence is identical, though reversed, from the suspened sequence. The `RESUME_POWER_ON` stage is performed first, with interrupts disabled. The driver is expected to power the device on. Interrupts are then enabled and the `RESUME_RESTORE_STATE` is performed, and the driver is expected to restore device state and free memory that was previously allocated.

A driver may use the `struct device_pm::saved_state` field to store a pointer to device state when the device is powered down.

Power Dependencies

Devices that are children of other devices (e.g. devices behind a PCI bridge) depend on their parent devices to be powered up to either provide power to them and/or provide I/O transactions.

The system must respect the power dependen-

cies of devices and must not attempt to power down a device which another device depends on being on. Put another way, all children devices must be powered down before their parent can be powered down. Conversely, the parent device must be powered up before any children devices may be accessed.

Expressing this type of dependency is simple, since it is easy to determine whether or not a device has any children or not. But, there are more interesting power dependencies that are more difficult to express.

On a PCI Hotplug system, the hotplug controller that controls power to a range of slots may reside on the primary PCI bus. However, the slots it controls may reside behind a PCI-PCI bridge that is a peer of the hotplug controller. The devices in the slots depend on the hotplug controller being on to operate, but it is not the devices' parent. There are similar transversal relationships on some embedded platforms in which some I/O controller resides near the system root that some PCI devices, several layers deep, may depend on to communicate properly.

Both types of power dependencies are represented using the `struct device_pm::depend` field. Implicit dependencies, like parent-child relationships, are handled by the depend count being incremented when a child is registered with the PM core. When that child device is powered down or removed, its parent's depend count is decremented. Only when a device's depend count is 0 may it be powered down.

Explicit power dependencies can be imposed on devices using

```
int device_pm_get(struct
    device *);
void device_pm_put(struct
     device *);
```

`device_pm_get()` will increment a device's dependency count, and `device_pm_put()` will decrement it. It is up to the driver to properly manage the dependency counts on device discovery, removal, and power management requests.

Disabling Power Management

There are circumstances in which a driver must refuse a power management request. This is usually because the driver author does not know the proper reinitialization sequence, or because the user is performing an uninterruptible operation like burning a CD.

It is valid for a driver to return an error from a `suspend()` method call. For example, a driver may know a priori that it can't handle the request. This works to the system's benefit, since the PM core can check if any devices have disabled power management before starting a suspend transition.

To disable or enable power management, a device may call

```
int device_pm_disable(struct
    device *);
void device_pm_enable(struct
     device *);
```

The former increments the `struct device_pm::disable` count, and the lattr decrements it. If the count is positive, system power management will be disabled completely, and device power management on that device.

These calls should be used judiciously, since they have a global impact on system power management.

3 System Power Management

System power management (SPM) is the process of placing the entire system into a low-power state. In a low-power state, the system is consuming a small, but minimal, amount of power, yet maintaining a relatively low response latency to the user. The exact amount of power and response latency depends on the state the system is in.

Power States

The states a system can enter are dependent on the underlying platform, and differ across architectures and even generations of the same architecture. There tend to be three states that are found on most archtitectures that support a form of SPM, though. The kernel explicitly supports these states—Standby, Suspend, and Hibernate, and provides a mechnanism for a platform driver (an architectural port of the kernel) to define new states.

```
typedef enum {
    POWER_ON         = 0,
    POWER_STANDBY    = 0x01,
    POWER_SUSPEND    = 0x02,
    POWER_HIBERNATE = 0x04,
} pm_system_state_t;
```

Standby is a low-latency power state that is sometimes referred to as "power-on suspend." In this state, the system conserves power by placing the CPU in a halt state and the devices in the D1 state. The power savings are not significant, but the response latency is minimal—typically less than 1 second.

Suspend is also commonly known as "suspend-to-RAM." In this state, all devices are placed in the D3 state and the entire system, except main memory, is expected to maintain power. Memory is placed in self-refresh mode, so its contents are not lost. Response latency is higher than Standby, yet still very low—between 3-5 seconds.

Hibernate conserves the most power by turning off the entire system, after saving state to a persistant medium, usually a disk. All devices are powered off unconditionally. The response latency is the highest—about 30 seconds—but still quicker than performing a full boot sequence.

Most platforms support these states, though some platforms may support other states or have requirements that don't match the assumptions above. For example, some PPC laptops support Suspend, but because of a lack of documentation, the video devices cannot be fully reinitialized and hence may not enter the D3 state. The hardware will supply enough power to devices for them to stay in the D2 state, which the drivers are capable of recovering from.

Instead of cluttering the code with a lot of conditional policy to determine the correct state for devices to enter, the PM subsystem abstracts system state information into dynamically registered objects.

```
    struct pm_state {
    struct pm_driver  * drv;
    pm_system_state_t   sys;
    pm_device_state_t   dev;
    struct kobject      kobj;
};
```

The drv field is a pointer to the platform-specific object configured to handle the power state. The sys field is the low-level power state that the system will enter. The dev field is the lowest power state that devices may enter. The kobj field is the generic object for managing an instance's lifetime.

The kernel defines default power state objects representing the assumptions above:

```
struct pm_state pm_state_standby;
struct pm_state pm_state_suspend;
struct pm_state pm_state_hibernate;
```

Platform drivers may also define and register additional power states that they support using:

```
int
pm_state_register(struct
                  pm_state *);
void
pm_state_unregister(struct
                    pm_state *);
```

The PM sysfs Interface

The PM infrastructure registers a top-level subsystem with the kobject core, which provides the `/sys/power/` directory in sysfs. By default, there is one file in the directory `/sys/power/state`.

Reading from this file displays the states that are currently registered with the system; e.g.:

```
# cat /sys/power/state
standby suspend hibernate
```

By writing the name of a state to this file, the system will perform a power state transition, which is described next.

Each power state that is registered receives a directory in `/sys/power/`, and three attribute files:

```
# tree /sys/power/suspend/
/sys/power/suspend/
|-- devices
|-- driver
‘-- system
```

The ‘devices’ file and the ‘system’ file describe which power state the devices in the computer and the state the computer itself are to enter, respectively. The ‘driver’ file displays which low-level platform PM driver is configured to handle the power transition. Writing to this file sets the driver internally.

Power Management Platform Drivers

The process of transitioning the OS into a low-power state is largely platform-agnostic. However, the low-level mechanism for actually transitioning the hardware to a low-power state is very platform specific, and even dependent on the generation of the hardware.

On some platforms, there may be multiple ways to enter a low-power state, presenting a policy decision for the user to make. Note this arises usually only in choosing whether to enter a minimal power state during a Hibernation transition, or turning the system completely off.

To cope with these variations, the PM core defines a simple driver model:

```
struct pm_driver {
    u32 states;
    int (*prepare)(u32 state);
    int (*save)   (u32 state);
    int (*sleep)  (u32 state);
    int (*restore)(u32 state);
    int (*cleanup)(u32 state);
    struct kobject  kobj;
};

int
pm_driver_register(struct
                   pm_driver *);

void
pm_driver_unregister(struct
                     pm_driver *);
```

The states field of `struct pm_driver` is a logical or of the states the driver supports. The methods are platform-specific calls that the PM

core executes during a power state transition. They are designed to perform the following:

prepare — Verify that the platform can enter the requested state and perform any necessary preparation for entering the state.

save — Save low-level state of the platform and the CPU(s).

sleep — Enter the requested state.

restore — Restore low-level register state of the platform and CPU(s).

cleanup — Perform any necessary actions to leave the sleep state.

A platform should intialize and register a driver on startup:

```
struct pm_driver acpi_pm_driver = {
  .states =  (POWER_STANDBY |
              POWER_SUSPEND  |
              POWER_HIBERNATE),
  .prepare = acpi_enter_sleep_state_prep,
  .sleep   = acpi_pm_sleep,
  .cleanup = acpi_leave_sleep_state,
  .kobj    = { .name = "acpi" },
};

static int __init acpi_sleep_init(void) {
    return pm_driver_register(
           &acpi_pm_driver);
}
```

Each registered PM driver receives a directory in sysfs in `/sys/power/`. Each driver receives one default attribute file named `states`, which displays the power states the driver supports. This file is not writable by userspace.

```
# tree /sys/power/acpi/
/sys/power/acpi/
'-- states
# cat /sys/power/acpi/states
standby suspend hibernate
```

Platform drivers may define and export their own attributes.

```
struct pm_attribute {
 struct attribute  attr;
 ssize_t (*show)(struct pm_driver *,
                 char *);
 ssize_t (*store)(struct pm_driver *,
                  const char *,
                  size_t);
};

int pm_attribute_create(
  struct pm_driver *,
  struct pm_attribute*);

void pm_attribute_remove(
  struct pm_driver *,
  struct pm_attribute *);
```

The semantics for `pm_driver` attributes follow the same semantics as other sysfs attributes. Please see the kernel sysfs documentation for more information.

Power State Transitions

Transitioning the system to a low-power state is, unfortunately, not as simple as telling the platform to enter the requested low power state. The file `drivers/power/suspend.c` contains the entire sequence, and should be used as reference material for the official process. A synopsis is provided here.

The first step is to verify that the system can enter the power state. The PM core must have a driver that supports the requested state, the driver must return success from its `prepare()` method, and the driver core must return success from `device_pm_check()`. Next, the PM core quiesces the running system by disabling preemption and 'freezing' all processes.

Next, system state is saved by calling `device_suspend()` to save device state,

and the driver's `save()` method to save low-level system state. If we're entering a variant of the Hibernation state, the contents of memory must be saved to a persistant medium. `pm_hibernate_save()` is called to perform this, which is described in the section Hibernation.

Once state is saved, the PM core disables interrupts and calls `device_power_down()` to place each device in the specified low power state. Finally, it calls the driver's `sleep()` method to transition the system to the low-power state.

The resume sequence has two variants, depending on whether the system is returning from a Hibernation state or not. If it is not, the platform is responsible for returning execution to the correct place (after the return from the driver's `sleep()` method). This may be a function of the processor, the firmware, or the low-level platform driver.

If we're returning from Hibernation, the system detects it during a boot process in the function `pm_resume()`. `pm_resume()` is a late_initcall, which means it is called after most subsystems and drivers have been registered and initialized, including all non-modular PM drivers. It calls `pm_hibernate_load()`, which is responsible for attempting to read, load, and restore a saved memory image. Doing this replaces the currently running system with a saved one, and execution returns to after the call to `pm_hibernate_store()`.

One way or another, the PM core proceeds to power on all devices and restore interrupts. The driver's `restore()` method is called to restore low-level system state, and `device_pm_resume()` is called to restore device context. Finally, the driver's `cleanup()` method is called, processes are 'thawed,' and preemption is reenabled.

A suspend transition is triggered by writing the requested state to the sysfs file `/sys/power/state`. Once the complete transition is complete, execution will return to the process that wrote the value.

4 Hibernation

This section describes the Hibernate power state; specifically the process the PM core uses to save memory to a persistant medium, and the model for implementing a low-level backend driver to read and write saved state from a specific medium.

As mentioned in the previous section, Hibernate is a low-power state in which system memory state is saved to a persistant medium before the system is powered off and restored during the system boot sequence.

Hibernate is the only low-power state that can be used in the absence of any platform support for power management. Instead of entering a low-power state, the configured PM driver may simply turn the system off. This mechanism provides perfect power savings (by not consuming any), and can be used to work around broken power management firmware or hardware. The PM core registers a default platform driver that supplies this mechanism. It is named 'shutdown' and supports the Hibernate state only.

Hibernation can also add value to situations which would otherwise ignore standard power management concepts. For example, system state can be saved and restored should a battery (either in a laptop or a UPS) become critcally low. Or, system state could be saved when the kernel Oops'd or hit a `BUG()`. The system could be rebooted and the state examined later.

Hibernation Backend Drivers

Hibernate is commonly referred to as "suspend-to-disk," implying that the medium that system state is saved to is a physical disk. This assumption does not offer the possibility that another type of media may be used to capture state, nor does it make the distinction of how the state is stored on disk, since it could theoretically be stored on a dedicated partition, in free swap space, or in a regular file on an arbitrary filesystem.

The PM subsystem offers the ability to configure a variable medium type to save state to.

```
struct pm_backend {
    int     (*open) (void);
    void    (*close)(void);
    int     (*read_image)(void);
    int     (*write_image)(void);
    struct kobject  kobj;
};

int pm_backend_register(struct p
    m_backend *);
void pm_backend_unregister(struct
     pm_backend *);
```

The PM core provides a default backend driver named `pmdisk` that uses a dedicated partition type to save state. The internals of pmdisk are discussed later.

Backend drivers are registered as children of the Hibernate `pm_state` object, and are represented by directories in sysfs.

They may also define and export attributes using the following interface:

```
struct pm_backend_attr {
 struct attribute attr;
 ssize_t (*show)(struct pm_backend *,
                 char *);

 ssize_t (*store)(struct pm_backend *,
                  const char *,
                  size_t);
};

int pm_backend_attr_create(
  struct pm_backend *,
  struct pm_backend_attr *);

void pm_backend_attr_remove(
  struct pm_backend *,
  struct pm_backend_attr *);
```

Snapshotting Memory

The Hibernate core 'snapshots' system memory by indexing and copying every active page in the system. Once a snapshot is complete, the saved image and index is passed to the backend driver to store persistantly.

The snapshot process has one critical requirement: that at least half of memory be free. This imposes a strict limitation on the use of the current Hibernate implementation during periods of high memory usage. However, this design decision simplifies the requirements of the implementaion itself.

The snapshot sequence is a three-step process. First, all of the active pages in the system are indexed, enough new pages are allocated to clone these pages, then each page is copied into its clone, or "shadow."

Active pages are detected by iterating over each page frame number (`'pfn'`) in the system and determining whether we should save it or not. A page's saveability is initially determined by whether or not the `PageNosave` bit is set, and then whether the page is free or not. Reserved pages may not be saveable, depending on whether they exist in the `'__nosave'` data section.

Pages marked `Nosave` or declared in the `__nosave` section (with the `'__nosavedata'` suffix) are volatile

data and variables internal to the Hibernate core. They are used and modified during the snapshot process, and are not saved.

Saveable pages are indexed in page-sized arrays called pm_chapters:

```
#define PG_PER_CHAPT \
    (PAGE_SIZE / sizeof(pgoff_t))

struct pm_chapter {
    pgoff_t c_pages[PG_PER_CHAPT];
};
```

pm_chapters are dynamically allocated based on the number of saveable pages in the system. The addresses of the allocated chapters are stored in another page-sized array, called a pm_volume:

```
#define CHAPT_PER_VOL \
    (PAGE_SIZE / \
    sizeof(struct pm_chapter *))

struct pm_volume {
    struct pm_chapter *
      v_chapters[CHAPT_PER_VOL];
};
```

There are two static pm_volumes in the Hibernate core—one for the memory index (pm_mem_index), and one for the snapshot (pm_mem_shadow). This imposes an upper limit on the amount of memory that can be snapshotted by the Hibernate core:

```
CHAPT_PER_VOL * PG_PER_CHAPT * PAGE_SIZE / 2
```

is the number of bytes that can be saved, assuming half of memory must be free to store the snapshot. On a 32-bit x86 machine with 4K-sized pages, this works out to be:

```
1024 * 1024 * 4096 / 2
= 2,147,483,648 bytes
= 2 GB
```

which is more than enough, since accessing memory above 1GB requires 4M-sized pages.

After memory has been indexed, but before it has been copied, the contents of pm_mem_index and pm_mem_shadow are copied to pm_mem_clone and pm_shadow_clone. The latter are also statically allocated objects, but are not declared "nosave." The purpose of the clones is to save the addresses of the dynamically allocated chapter pages so we can free them once the saved image has been restored.

At this stage, the Hibernate core calls a required architecture- specific function:

```
int pm_arch_hibernate(
     pm_system_state_t state);
```

The state parameter should be set to POWER_HIBERNATE. This call is responsible for saving low-level register state and calling pm_hibernate_save(), which copies each indexed page in pm_mem_index to its corresponding page in pm_mem_shadow.

Restoring Memory

During a resume sequence, the Hibernate core calls the backend's open() method, which is responsible for setting pm_num_pages, which the Hibernate core will use to pre-allocate pm_mem_index and pm_mem_shadow.

The backend's read_image() method is called, which populates pm_mem_index with the target location of each saved page, and pm_mem_shadow, which contains the saved pages.

The saved image will replace the memory on a different running system. The pages that have been allocated to store the saved image populated from the backend may conflict with pages

in the saved image that are to be restored. The Hibernate backend must guarantee that none of the pages currently pointed to by `pm_mem_shadow` conflict with the pages indexed by `pm_mem_index`. To do this, it loops through each page address in `pm_mem_shadow` and compares them with each page address in `pm_mem_index`. If any matches are found, a new page is allocated and the contents copied.

To replace memory, the Hibernate core calls

```
pm_arch_hibernate(POWER_ON);
```

The architecture is responsible for iterating over the pages in `pm_mem_shadow` and copying each one to its destination, as indexed in `pm_mem_index`. It is also responsible for restoring low-level register state once memory has been replaced.

This burden is placed on the architecture so it can implement a replacement algorithm without using the stack for variable storage. The saved memory image contains the saved stack, while the current stack pointer register will point to a location on the stack in the memory being replaced. These will likely not match and cause the system to crash very quickly.

Once the memory image is restored, the architecture must restore register context to get the stack pointer pointing to the right place. This is the reason that the same function is called to both save and restore the low-level registers.

Returning from `pm_arch_hibernate()` once memory has been replaced will restore execution to the point in `hibernate_write()` where `pm_arch_hibernate()` was called, in the saving sequence. To detect this, the Hibernate core declares:

```
static in_suspend __nosavedata = 0;
```

and sets to it one during the save path. Since it's not saved, it will be 0 during the restore path, allowing the Hibernate core to behave appropriately. The cloned volumes are copied back into `pm_mem_index` and `pm_mem_shadow`, and the dynamically allocated pages are freed.

Backend Driver Semantics

The Hibernate core calls the backend driver's `open()` method before any Hibernate operation. It is the backend's responsibility to verify the existence of the media and to open any necessary communication channels to it. The backend driver is responsible for reading image metadata from the medium and setting `pm_num_pages` to the number of saved pages if a saved image exists. The Hibernate core will use this value to pre-allocate storage for the saved pages.

It may also use this opportunity to verify there is enough free space on the device. The maximum requirement is the total amount of memory in the system, as indicated by:

```
num_physpages * PAGE_SIZE
```

This check is optional at this stage, since the size of the saved memory image may be much smaller than this, and may fit on a device with less free space than the total size of memory.

When the Hibernate core is done, it will call the backend's `close()` method. The backend is responsible for closing any communication channels to the storage medium and freeing any memory it had allocated.

After the Hibernate core has shadowed memory, it calls the backend's `write_image()` method. It does not pass any parameters. `pm_mem_index` and `pm_mem_shadow` must be used directly. The backend must saved each

page pointed to in each chapter of `pm_mem_shadow`. It must also save each chapter page of `pm_mem_index`. The exact format in which these are saved are up to the driver.

When restoring a memory image, after the Hibernate core has allocated storage for the saved memory, the backend's `read_image()` method is called. `pm_mem_index` contains enough allocated chapters to store the saved chapters and `pm_mem_shadow` contains enough allocated chapters and pages to store all of the saved pages. The backend must populate all of these.

pmdisk

`pmdisk` is a simple hibernate backend driver. It uses a dedicated partition with a custom format for storing system state. Internally, pmdisk uses the bio layer to read and write pages directly to/from the disk.

A pmdisk partition may be created using a utility called `pmdisk`. It simply writes a pmdisk header to a partition, which is defined as:

```
#define PM_HIBERNATE_SIG "PMHibernate"
#define PM_HIBERNATE_VER 1

#define PM_UNUSED_SPACE \
   (PAGE_SIZE - (4 * \
   sizeof(unsigned long) + 16))

struct pmdisk_header {
  char  h_unused[PM_UNUSED_SPACE];
  unsigned long h_version;
  unsigned long h_chksum;
  unsigned long h_pages;
  unsigned long h_chapters;
  char          h_sig[16];
} __attribute__((packed));
```

Internally, the pmdisk backend driver reads the header from the first page of the configured partition when its `open()` method is called. It verifies that it is a pmdisk partition, and sets `pm_num_pages` if there is an image stored on the disk.

On a `close()` call, pmdisk set the `h_pages`, `h_chapters`, and `h_chksum` fields of the header and writes it to the first page on the disk. Note that on a memory restore operation, `pm_num_pages` will be 0, signifying the memory image on the disk is no longer valid.

A saved memory image on a pmdisk partition is layed out like:

0:	pmdisk header
1 to `Nc`	Saved chapters of `pm_mem_index`
`Nc` to `Np`	Saved pages from `pm_mem_shadow`

On a `write_image()` call, pmdisk will first initialize an internal checksum variable.It will then write each chapter from `pm_mem_index` to disk, then each page from `pm_mem_shadow` to disk. As it writes each page, it will pass it to a checksum function. The checksum function is simple and definitely not cryptographically secure. But, it does provide an easy verification that an image on disk is valid.

On a `read_image()` call, pmdisk reads each chapter into `pm_mem_index` and each page into `pm_mem_shadow`. As it reads each page, it checksums them. Once all pages have been read, it compares the current checksum with the `h_chksum` field of the header. It returns success only if they match.

The internal pmdisk exports a sysfs attribute file named 'dev' which userspace must use to tell the kernel of the correct pmdisk partition to use. There is currently no way for pmdisk to automatically detect any valid partitions in the system.

The value that userspace must write to `dev` is a 16-bit `dev_t` value in hexadecimal format containing the major/minor number pair of the device to use. This format is not favored, but

is the only current method for obtaining a reference to a specific block device at the time of writing. This interface will change in the future.

5 Other Power Management Options

So far, a lot of talk has been dedicated to describing the internals of the new power management subsystem, but little has been given to describe how the new infrastructure interacts with current power management options. This section describes those relationships, and although it focuses on options specific to ia32 platforms, the relationships should be extendable to other platforms.

ACPI

In terms of system power management, fits nicely into the new PM infrastructure. It behaves as a PM driver, and provides platform-specific hooks to transition the system into a low-power state. At the basic SPM level, this is all that is required, though ACPI offers a potentially much more powerful solution, since it it exposes more intimate knowledge of the platform power requirements than has ever been available on ia32 platforms (e.g. response latencies, power consumption etc.). Exploiting this knowledge is up to the ACPI platform driver to expose these attributes via sysfs.

ACPI offers similar potential for device power management. Devices that appear in the firmware's DSDT (Differentiated System Description Table) may expose a very fine-grained level of detail about the devices' power requirements and capabilities.

ACPI also stress the capabilities of device Performance States. A performance state is a power state that describes a trade-off between the capabilities of a device against the power consumption of the device. In each performance state that a device supports, the device is fully running, but different functional hardware units may be powered off to conserve power. The driver model does not explicitly recognize performance states, though the new PM extensions to the driver model provide a framework that could easily be extended to recognize performance states.

APM

APM power management does not appear on very many new systems, but the current Linux installed base includes a large number of APM-capable computers. The new PM model was not developed with APM, or any firmware-driven PM model, in mind. However, care was taken to ensure that it conceptually made sense to use such mechanisms as low-level platform drivers for the PM model. No work has been done, however, to convert APM to act as a PM driver for the new model.

pm infratructure

The original PM infrastucture was developed by Andrew Henroid and was very groundbreaking, since nothing like it had been done for the Linux kernel before. It exists in its entirety in:

```
kernel/pm.c
include/linux/pm.h
```

The general idea is that drivers can declare and register an object with the pm infrastructure that is accessed during a power state transition. The idea is very similar to what we have now, though the registration now is implicit when a device is registered with the system. And, based on the implementation, we can guarantee

that each device is notified in proper ancestral order, which the old model cannot do.

Because the new model is far superior the old-style pm infrastructure, it is declared deprecated. All drivers that implement pm callbacks should be converted to use the hooks provided by the new driver model.

swsusp

swsusp is a mechanism for doing suspend-to-disk by saving kernel state to unused swap space. It was also a ground-breaking feature, as it was the first true suspend-to-disk implementation for Linux. There are some questionable characteristics of swsusp that many people have that the maintainers of swsusp counter are frivilous concerns, and it currently exists as an alternative to the new PM model. However, I've revoked any philosophical issues with swsusp. It can be, and should be ported to be, used as a backend driver for the generic Hibernate mechanism. The current code base could be reduced to a fraction of its current complexity.

6 Resources

The current power management kernel tree can be found in the BitKeeper repository:

```
bk://devloper.osdl.org/
linux-2.5-power
```

Information about the Linux power managment infrastructure, including GNU diffs, documentation and utilities like pmdisk can be found at

```
http://developer.osdl.org/
~mochel/power/
```

General OSDL developer resources can be found at:

```
http://developer.osdl.org/
```

7 Acknowledgements

Many people have contributed to this document, both explicitly and implicitly. First, Linus deserves a mention for encouraging me look at implementing ACPI suspend-to-ram as my first kernel project. Andy Grover and Paul Diefenbaugh of Intel for many things—contributing ACPI to the kernel, for talking with me, for always arguing with and motivating me internally to do things better, and for pushing me over the edge to write the finest OS driver model in existence. Andy Henroid for writing the first open-source power management model and providing a great base—despite its shortcomings—to learn and build from. Pavel Machek for constantly providing code and being energetic about the project. All the swsusp people for doing it in the first place and keeping it up, no matter how much I gripe about it.

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*