The sysfs Filesystem

Patrick Mochel
mochel@digitalimplant.org

Abstract

sysfs is a feature of the Linux 2.6 kernel that allows kernel code to export information to user processes via an in-memory filesystem. The organization of the filesystem directory hierarchy is strict, and based the internal organization of kernel data structures. The files that are created in the filesystem are (mostly) ASCII files with (usually) one value per file. These features ensure that the information exported is accurate and easily accessible, making sysfs one of the most intuitive and useful features of the 2.6 kernel.

Introduction

sysfs is a mechanism for representing kernel objects, their attributes, and their relationships with each other. It provides two components: a kernel programming interface for exporting these items via sysfs, and a user interface to view and manipulate these items that maps back to the kernel objects which they represent. The table below shows the mapping between internel (kernel) constructs and their external (userspace) sysfs mappings.

Internal	**External**
Kernel Objects	Directories
Object Attributes	Regular Files
Object Relationships	Symbolic Links

sysfs is a core piece of kernel infrastructure, which means that it provides a relatively simple interface to perform a simple task. Rarely is the code overly complicated, or the descriptions obtuse. However, like many core pieces of infrastructure, it can get a bit too abstract and far removed to keep track of. To help alleviate that, this paper takes a gradual approach to sysfs before getting to the nitty-gritty details.

First, a short but touching history describes its origins. Then crucial information about mounting and accessing sysfs is included. Next, the directory organization and layout of subsystems in sysfs is described. This provides enough information for a user to understand the organization and content of the information that is exported through sysfs, though for reasons of time and space constraints, not every object and its attributes are described.

The primary goal of this paper is to provide a technical overview of the internal sysfs interface—the data structures and the functions that are used to export kernel constructs to userspace. It describes the functions among the three concepts mentioned above—Kernel Objects, Object Attributes, and Object Relationships—and dedicates a section to each one. It also provides a section for each of the two additional regular file interfaces created to simplify some common operations—Attribute Groups and Binary Attributes.

sysfs is a conduit of information between the kernel and user space. There are many opportunities for user space applications to leverage this information. Some existing uses are the ability to I/O Scheduler parameters and the udev program. The final section describes a sampling of the current applications that use sysfs and attempts to provide enough inspiration to spawn more development in this area.

Because it is a simple and mostly abstract interface, much time can be spent describing its interactions with each subsystem that uses it. This is especially true for the kobject and driver models, which are both new features of the 2.6 kernel and heavily intertwined with sysfs. It would be impossible to do those topics justice in such a medium and are left as subjects for other documents. Readers still curious in these and related topics are encouraged to read [4].

1 The History of sysfs

sysfs is an in-memory filesystem that was originally based on ramfs. ramfs was written around the time the 2.4.0 kernel was being stabilized. It was an exercise in elegance, as it showed just how easy it was to write a simple filesystem using the then-new VFS layer. Because of its simplicity and use of the VFS, it provided a good base from which to derive other in-memory based filesystems.

sysfs was originally called *ddfs* (Device Driver Filesystem) and was written to debug the new driver model as it was being written. That debug code had originally used procfs to export a device tree, but under strict urging from Linus Torvalds, it was converted to use a new filesystem based on ramfs.

By the time the new driver model was merged into the kernel around 2.5.1, it had changed names to *driverfs* to be a little more descriptive. During the next year of 2.5 development, the infrastructural capabilities of the driver model and driverfs began to prove useful to other subsystems. kobjects were developed to provide a central object management mechanism and driverfs was converted to sysfs to represent its subsystem agnosticism.

```
mount -t sysfs sysfs /sys
```

Table 1: A sysfs mount command

```
sysfs  /sys  sysfs  noauto  0 0
```

Table 2: A sysfs entry in /etc/fstab

2 Mounting sysfs

sysfs can be mounted from userspace just like any other memory-based filesystem. The command for doing so is listed in Table 1.

sysfs can also be mounted automatically on boot using the file `/etc/fstab`. Most distributions that support the 2.6 kernel have entries for sysfs in `/etc/fstab`. An example entry is shown in Table 2.

Note that the directory that sysfs is mounted on: `/sys`. That is the de facto standard location for the sysfs mount point. This was adopted without objection by every major distribution.

3 Navigating sysfs

Since sysfs is simply a collection of directories, files, and symbolic links, it can be navigated and manipulated using simple shell utilities. The author recommends the `tree(1)` utility. It was an invaluable aide during the development of the core kernel object infrastructure.

```
/sys/
|-- block
|-- bus
|-- class
|-- devices
|-- firmware
|-- module
‘-- power
```

Table 3: Top level sysfs directories

At the top level of the sysfs mount point are a number of directories. These directories represent the major subsystems that are registered with sysfs. At the time of publication, this consisted of the directories listed in Table 3. These directories are created at system startup when the subsystems register themselves with the kobject core. After they are initialized, they begin to discover objects, which are registered within their respective directories.

The method by which objects register with sysfs and how directores are created is explained later in the paper. In the meantime, the curious are encouraged to meander on their own through the sysfs hierarchy, and the meaning of each subsystem and their contents follows now.

3.1 block

The `block` directory contains subdirectories for each block device that has been discovered in the system. In each block device's directory are attributes that describe many things, including the size of the device and the dev_t number that it maps to. There is a symbolic link that points to the physical device that the block device maps to (in the physical device tree, which is explained later). And, there is a directory that exposes an interface to the I/O scheduler. This interface provides some statistics about about the device request queue and some tunable features that a user or administrator can use to optimize performance, including the ability to dyanmically change the I/O scheduler to use.

Each partition of each block device is represented as a subdirectory of the block device. Included in these directories are read-only attributes about the partitions.

```
bus/
|-- ide
|-- pci
|-- scsi
‘-- usb
```

Table 4: The bus directory

3.2 bus

The `bus` directory contains subdirectories for each physical bus type that has support registered in the kernel (either statically compiled or loaded via a module). Partial output is listed in Table 4.

Each bus type that is represented has two subdirectories: `devices` and `drivers`. The `devices` directory contains a flat listing of every device discovered on that type of bus in the entire system. The devices listed are actually symbolic links that point to the device's directory in the global device tree. An example listing is shown in Table 5.

The `drivers` directory contains directories for each device driver that has been registered with the bus type. Within each of the drivers' directories are attributes that allow viewing and manipulation of driver parameters, and symbolic links that point to the physical devices (in the global device tree) that the driver is bound to.

```
bus/pci/devices/
|-- 0000:00:00.0 -> ../../../devices/pci0000:00/0000:00:00.0
|-- 0000:00:01.0 -> ../../../devices/pci0000:00/0000:00:01.0
|-- 0000:01:00.0 -> ../../../devices/pci0000:00/0000:00:01.0/0000:01:00.0
|-- 0000:02:00.0 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:00.0
|-- 0000:02:00.1 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:00.1
|-- 0000:02:01.0 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:01.0
'-- 0000:02:02.0 -> ../../../devices/pci0000:00/0000:00:1e.0/0000:02:02.0
```

Table 5: PCI devices represented in `bus/pci/devices/`

```
class/
|--  graphics
|--  input
|--  net
|--  printer
|--  scsi_device
|--  sound
'--  tty
```

Table 6: The class directory

3.3 class

The `class` directory contains representations of every device class that is registered with the kernel. A device class describes a functional type of device. Examples of classes are shown in Table 6.

Each device class contains subdirectories for each class object that has been allocated and registered with that device class. For most of class device objects, their directories contain symbolic links to the device and driver directories (in the global device hierarchy and the bus hierarchy respectively) that are associated with that class object.

Note that there is not necessarily a 1:1 mapping between class objects and physical devices; a physical device may contain multiple class objects that perform a different logical function. For example, a physical mouse device might map to a kernel mouse object, as well as a generic "input event" device and possibly a "input debug" device.

Each class and class object may contain attributes exposing parameters that describe or control the class object. The contents and format, though, are completely class dependent and depend on the support present in one's kernel.

3.4 devices

The `devices` directory contains the global device hierarchy. This contains every physical device that has been discovered by the bus types registered with the kernel. It represents them in an ancestrally correct way—each device is shown as a subordinate device of the device that it is physically (electrically) subordinate to.

There are two types of devices that are exceptions to this representation: platform devices and system devices. Platform devices are peripheral devices that are inherent to a particular platform. They usually have some I/O ports, or MMIO, that exists at a known, fixed location. Examples of platform devices are legacy x86 devices like a serial controller or a floppy controller, or the embedded devices of a SoC solution.

System devices are non-peripheral devices that are integral components of the system. In many ways, they are nothing like any other device. They may have some hardware register access for configuration, but do not have the capability to transfer data. They usually do not have drivers which can be bound to them. But, at least for those represented through sysfs, have some architecture-specific code that configures them and treats them enough as objects to export them. Examples of system devices are CPUs, APICs, and timers.

3.5 firmware

The `firmware` directory contains interfaces for viewing and manipulating firmware-specific objects and attributes. In this case, 'firmware' refers to the platform-specific code that is executed on system power-on, like the x86 BIOS, OpenFirmware on PPC platforms, and EFI on ia64 platforms.

Each directory contains a set of objects and attributes that is specific to the firmware "driver in the kernel." For example, in the case of ACPI, every object found in the ACPI DSDT table is listed in `firmware/acpi/namespace/` directory.

3.6 module

The `module` directory contains subdirectories for each module that is loaded into the kernel. The name of each directory is the name of the module—both the name of the module object file and the internal name of the module. Every module is represented here, regardless of the subsystem it registers an object with. Note that the kernel has a single global namespace for all modules.

Within each module directory is a subdirectory called `sections`. This subdirectory contains attributes about the module sections. This information is used for debugging and generally not very interesting.

Each module directory also contains at least one attribute: `refcnt`. This attributes displays the current reference count, or number of users, of the module. This is the same value in the fourth column of `lsmod(8)` output.

3.7 power

The `power` directory represents the underused power subsystem. It currently contains only two attributes: `disk` which controls the method by which the system will suspend to disk; and `state`, which allows a process to enter a low power state. Reading this file displays which states the system supports.

4 General Kernel Information

4.1 Code Organization

The code for sysfs resides in `fs/sysfs/` and its shared function prototypes are in `include/linux/sysfs.h`. It is relatively small (~2000 lines), but it is divided up among 9 files, including the shared header file. The organization of these files is listed below. The contents of each of these files is described in the next section.

- `include/linux/sysfs.h` - Shared header file containing function prototypes and data structure definitions.
- `fs/sysfs/sysfs.h` - Internal header file for sysfs. Contains function definitions shared locally among the sysfs source.
- `fs/sysfs/mount.c` - This contains the data structures, methods, and initialization functions necessary for interacting with the VFS layer.

- `fs/sysfs/inode.c` - This file contains internal functions shared among the sysfs source for allocating and freeing the core filesystem objects.
- `fs/sysfs/dir.c` - This file contains the externally visible sysfs interface responsible for creating and removing directories in the sysfs hierarchy.
- `fs/sysfs/file.c` - This file contains the externally visible sysfs interface responsible for creating and removing regular, ASCII files in the sysfs hiearchy.
- `fs/sysfs/group.c` - This file contains a set of externally-visible helpers that aide in the creation and deletion of multiple regular files at a time.
- `fs/sysfs/symlink.c` - This file contains the externally- visible interface responsible for creating and removing symlink in the sysfs hierarchy.
- `fs/sysfs/bin.c` - This file contains the externally visible sysfs interface responsible for creating and removing binary (non-ASCII) files.

4.2 Initialization

sysfs is initialized in `fs/sysfs/mount.c`, via the `sysfs_init` function. This function is called directly by the VFS initialization code. It must be called early, since many subsystems depend on sysfs being initialized to register objects with. This function is responsible for doing three things.

- **Creating a `kmem_cache`.** This cache is used for the allocation of `sysfs_dirent` objects. These are discussed in a later section.
- **Registering with the VFS.** `register_filesystem()` is called with the `sysfs_fs_type` object. This sets up the appropriate super block methods and adds a filesystem with the name `sysfs`.
- **Mounts itself internally.** This is done to ensure that it is always available for other kernel code to use, even early in the boot process, instead of depending on user interaction to explicitly mount it.

Once these actions complete, sysfs is fully functional and ready to use by all internal code.

4.3 Configuration

sysfs is compiled into the kernel by default. It is dependent on the configuration option `CONFIG_SYSFS`. `CONFIG_SYSFS` is only visible if the `CONFIG_EMBEDDED` option is set, which provides many options for configuring the kernel for size-constrained envrionments. In general, it is considered a good idea to leave sysfs configured in a custom-compiled kernel. Many tools currently do, and probably will in the future, depend on sysfs being present in the system.

4.4 Licensing

The sysfs code is licensed under the GPLv2. While most of it is now original, it did originate as a clone of ramfs, which is licensed under the same terms. All of the externally-visible interfaces are original works, and are of course also licensed under the GPLv2.

The external interfaces are exported to modules, however only to GPL-compatible modules, using the macro `EXPORT_SYMBOL_GPL`. This is done for reasons of maintainability and derivability. sysfs is a core component of the kernel. Many subsystems rely on it, and while it is a stable piece of infrastructure, it occasionally must change. In order to develop the best possible modifications, it's imperative that all callers of sysfs interfaces be audited and updated in lock-step with any sysfs interface changes. By requiring that all users be licensed in a GPL manner, and hopefully merged into the kernel, the level of difficulty of an interface change can be greatly reduced.

Also, since sysfs was developed initially as an extension of the driver model and has gone through many iterations of evolution, it has a very explicit interaction with its users. To develop code that used sysfs but was not copied or derived from an existing in-kernel GPL-based user would be difficult, if not impossible. By requiring GPL-compatibility in the users of sysfs, this can be made explicit and help prevent falsification of derivability.

5 Kernel Interface Overview

The sysfs functions visible to kernel code are divided into three categories, based on the type of object they are exporting to userspace (and the type of object in the filesystem they create).

- Kernel Objects (Directories).
- Object Attributes (Regular Files).
- Object Relationships (Symbolic Links).

There are also two other sub-categories of exporting attributes that were developed to accomodate users that needed to export other files besides single, ASCII files. Both of these categories result in regular files being created in the filesystem.

- Attribute Groups
- Binary Files

The first parameter to all sysfs functions is the kobject (hereby referenced as `k`), which is being manipulated. The sysfs core assumes that this kobject will remain valid throughout the function; i.e., they will not be freed. The caller is always responsible for ensuring that any necessary locks that would modify the object are held across all calls into sysfs.

For almost every function (the exception being `sysfs_create_dir`), the sysfs core assumes that `k->dentry` is a pointer to a valid dentry that was previously allocated and initialized.

All sysfs function calls must be made from process context. They should also not be called with any spinlocks held, as many of them take semaphores directly and all call VFS functions which may also take semaphores and cause the process to sleep.

6 Kernel Objects

Kernel objects are exported as directories via sysfs. The functions for manipulating these directories are listed in Table 7.

`sysfs_create_dir` is the only sysfs function that does not rely on a directory having already been created in sysfs for the kobject (since it performs the crucial action of creating that directory). It does rely on the following parameters being valid:

- k->parent
- k->name

```
int sysfs_create_dir(struct kobject * k);

void sysfs_remove_dir(struct kobject * k);

int sysfs_rename_dir(struct kobject *, const char *new_name);
```

Table 7: Functions for manipulating sysfs directories.

6.1 Creating Directories

These parameters control where the directory will be located and what it will be called. The location of the new directory is implied by the value of `k->parent`; it is created as a subdirectory of that. In all cases, the subsystem (not a low-level driver) will fill in that field with information it knows about the object when the object is registered with the subsystem. This provides a simple mechanism for creating a complete user-visible object tree that accurately represents the internal object tree within the kernel.

It is possible to call `sysfs_create_dir` without `k->parent` set; it will simply create a directory at the very top level of the sysfs filesystem. This should be avoided unless one is writing or porting a new top-level subsystem using the kobject/sysfs model.

When `sysfs_create_dir()` is called, a dentry (the object necessary for most VFS transactions) is allocated for the directory, and is placed in `k->dentry`. An inode is created, which makes a user-visible entity, and that is stored in the new dentry. sysfs fills in the `file_operations` for the new directory with a set of internal methods that exhibit standard behavior when called via the VFS system call interface. The return value is 0 on success and a negative errno code if an error occurs.

6.2 Removing Directories

`sysfs_remove_dir` will remove an object's directory. It will also remove any regular files that reside in the directory. This was an original feature of the filesystem to make it easier to use (so all code that created attributes for an object would not be required to be called when an object was removed). However, this feature has been a source of several race conditions throughout the years and should not be relied on in the hopes that it will one day be removed. All code that adds attributes to an object's directory should explicitly remove those attributes when the object is removed.

6.3 Renaming Directories

`sysfs_rename_dir` is used to give a directory a new name. When this function is called, sysfs will allocate a new dentry for the kobject and call the kobject routine to change the object's name. If the rename succeeds, this function will return 0. Otherwise, it will return a negative errno value specifying the error that occurred.

It is not possible at this time to move a sysfs directory from one parent to another.

7 Object Attributes

Attributes of objects can be exposed via sysfs as regular files using the `struct attribute`

```
  int sysfs_create_file(struct kobject *, const struct attribute *);

  void sysfs_remove_file(struct kobject *, const struct attribute *);

  int sysfs_update_file(struct kobject *, const struct attribute *);
```

Table 8: Functions for manipulating sysfs files

```
struct device_attribute {
  struct attribute        attr;
  ssize_t (*show)(struct device *dev, char *buf);
  ssize_t (*store)(struct device *dev, const char *buf, size_t count);
};

int device_create_file(struct device *device,
                       struct device_attribute *entry);
void device_remove_file(struct device *dev,
                        struct device_attribute *attr);
```

Table 10: A wrapper for `struct attribute` from the Driver Model

```
struct attribute {
    char             *name;
    struct module    *owner;
    mode_t           mode;
};
```

Table 9: The `struct attribute` data type

data type described in Table 9 and the functions listed in Table 8.

7.1 Creating Attributes

`sysfs_create_file()` uses the `name` field to determine the file name of the attribute and the `mode` field to set the UNIX file mode in the file's inode. The directory in which the file is created is determined by the location of the kobject that is passed in the first parameter.

7.2 Reference Counting and Modules

The `owner` field may be set by the the caller to point to the module in which the attribute code exists. This should **not** point to the module that owns the kobject. This is because attributes can be created and removed at any time. They do not need to be created when a kobject is registered; one may load a module with several attributes for objects of a particular type that are registered after the objects have been registered with their subsystem.

For example, network devices have a set of statistics that are exported as attributes via sysfs. This set of statistics attributes could reside in an external module that does not need to be loaded in order for the network devices to function properly. When it is loaded, the attributes contained within are created for every registered network device. This module could be unloaded at any time, removing the attributes from sysfs for each network device. In this case, the `module` field should point to the module that contains the network statistic attributes.

The `owner` field is used for reference counting when the attribute file is accessed. The file operations for attributes that the VFS calls are set by sysfs with internal functions. This allows sysfs to trap each access call and perform necessary actions, and it allows the actual methods that read and write attribute data to be greatly simplified.

When an attribute file is opened, sysfs increments the reference count of both the kobject represented by the directory where the attribute resides, and the module which contains the attribute code. The former operation guarantees that the kobject will not be freed while the attribute is being accessed. The latter guarantees that the code which is being executed will not be unloaded from the kernel and freed while the attribute is being accessed.

7.3 Wrappable Objects

One will notice that `struct attribute` does not actually contain the methods to read or write the attribute. sysfs does not specify the format or parameters of these functions. This was an explicit design decision to help ensure type safety in these functions, and to aid in simplifying the downstream methods.

Subsystems that use sysfs attributes create a new data type that encapsulates `struct attribute`, like in Table 10. By defining a wrapping data type and functions, downstream code is protected from the low-level details of sysfs and kobject semantics.

When an attribute is read or written, sysfs accesses a special data structure, through the kobject, called a kset. This contains the base operations for reading and writing attributes for kobjects of a particular type. These functions translate the kobject and attribute into higher level objects, which are then passed to the `show` and `store` methods described in Table 10. Again, this helps ensure type safety, because it guarantees that the downstream function receives a higher-level object that it use directly, without having to translate it.

Many programmers are inclined to cast between object types, which can lead to hard-to-find bugs if the position of the fields in a structure changes. By using helper functions within the kernel that perform an offset-based pointer subtraction to translate between object types, type safety can be guaranteed, regardless if the field locations may change. By centralizing the translation of objects in this manner, the code can be easier to audit in the event of change.

7.4 Reading and Writing Attributes

sysfs attempts to make reading and writing attributes as simple as possible. When an attribute is opened, a `PAGE_SIZE` buffer is allocated for transferring the data between the kernel and userspace. When an attribute is read, this buffer is passed to a downstream function (e.g., `struct device_attribute::show()` which is responsible for filling in the data and formatting it appropriately. This data is then copied to userspace.

When a value is written to a sysfs attribute file, the data is first copied to the kernel buffer, then it is passed to the downstream method, along with the size of the buffer in bytes. This method is responsible for parsing the data.

It is assumed that the data written to the buffer is in ASCII format. It is also implied that the size of the data written is less than one page in size. If the adage of having one value per file is followed, the data should be well under one page in size. Having only one value per file also eliminates the need for parsing complicated strings. Many bugs, especially in text parsing, are propogated throughout the kernel by copying and pasting code thought to be bug-free. By making it easy to export one value per file, sysfs eliminates the need for copy-and-paste development, and prevents these bugs from propagating.

7.5 Updating an attribute

If the data for an attribute changes, kernel code can notify a userspace process that may be waiting for updates by modifying the timestamp of the file using `sysfs_update_file()`. This function will also call dnotify, which some applications use to wait for modified files.

8 Object Relationships

A relationship between two objects can be expressed in sysfs by the use of a symbolic link. The functions for manipulating symbolic links in sysfs are shown in Table 11. A relationship within the kernel may simply be a pointer between two different objects. If both of these objects are represented in sysfs with directories, then a symbolic link can be created between them and prevent the addition of redundant information in both objects' directories.

When creating a symbolic link between two objects, the first argument is the kobject that is being linked *from*. This represents the directory in which the symlink will be created. The second argument is the kobject which is being linked *to*. This is the directory that the symlink will point to. The third argument is the name of the symlink that will appear in the filesystem.

To illustrate this, consider a PCI network device and driver. When the system boots, the PCI device is discovered and a sysfs directory is created for it, long before it is bound to a specific driver. At some later time, the network driver is loaded, which may or may not bind to any devices. This is a different object type than the physical PCI device represents, so a new directory is created for it.

Their association is illustrated in Table 12. Shown is the driver's directory in sysfs, which is named after the name of the driver module. This contains a symbolic link that points to the devices to which it is bound (in this case, just one). The name of the symbolic link and the target directory are the same, and based on the physical bus ID of the device.

9 Attribute Groups

The attribute group interface is a simplified interface for easily adding and removing a set of attributes with a single call. The `attribute_group` data structure and the functions defined for manipulating them are listed in Table 13.

An attribute group is simply an array of attributes to be added to an object, as represented by the `attrs` field. The `name` field is optional. If specified, sysfs will create a subdirectory of the object to store the attributes in the group. This can be a useful aide in organizing large numbers of attributes.

Attribute groups were created to make it easier to keep track of errors when registering multiple attributes at one time, and to make it more compelling to clean up all attributes that a piece of code may create for an object. Attributes can

```
int sysfs_create_link(struct kobject *kobj, struct kobject *target,
                      char *name);

void sysfs_remove_link(struct kobject *, char *name);
```

Table 11: Functions for manipulating symbolic links in sysfs

```
$ tree -d bus/pci/drivers/e1000/
bus/pci/drivers/e1000/
‘-- 0000:02:01.0 -> ../../../../devices/pci0000:00/0000:00:1e.0/0000:02:01.0
```

Table 12: An example of a symlink in sysfs.

be added and removed from the group without having to change the registration and unregistration functions.

When a group of attributes is added, the return value is noted for each one. If any one fails to be added (because of e.g. low memory conditions or duplicate attribute names), the previously added attributes of that group will be removed and the error code will be returned to the caller. This allows downstream code to retain a simple and elegant error handling mechanism, no matter how many attributes it creates for an object.

When an attribute group is removed, all of the attributes contained in it are removed. If a subdirectory was created to house the attributes, it is also removed.

Good examples of attribute groups and their uses can be found in the network device statistics code. Its sysfs interface is in the file `net/core/net-sysfs.c`.

10 Binary Attributes

Binary files are a special class of regular files that can be exported via sysfs using the data structure and functions listed in Table 14. They exist to export binary data structures that are either best left formatted and parsed in a more flexible environment, like userspace process context because they have a known and standard format (e.g., PCI Configuration Space Registers); or because their use is strictly in binary format (e.g., binary firmware images).

The use of binary files is akin to the procfs interface, though sysfs still traps the read and write methods of the VFS before it calls the methods in `struct bin_attribute`. It allows more control over the format of the data, but is more difficult to manage. In general, if there is a choice over which interface to use—regular attributes or binary attributes, there should be no compelling reasons to use binary attributes. They should only be used for specific purposes.

11 Current sysfs Users

The number of applications that use sysfs directly are few. It already provides a substantial amount of useful information in an organized format, so the need for utilities to extract and parse data is minimal. However, there are a few users of sysfs, and the infrastructure to support more is already in place.

```
struct attribute_group {
    char                *name;
    struct attribute   **attrs;
};

int sysfs_create_group(struct kobject *,
                       const struct attribute_group *);
void sysfs_remove_group(struct kobject *,
                        const struct attribute_group *);
```

Table 13: Attribute Groups

```
struct bin_attribute {
    struct attribute        attr;
    size_t                  size;
    void                    *private;
    ssize_t (*read)(struct kobject *, char *, loff_t, size_t);
    ssize_t (*write)(struct kobject *, char *, loff_t, size_t);
    int (*mmap)(struct kobject *, struct bin_attribute *attr,
                struct vm_area_struct *vma);
};

int sysfs_create_bin_file(struct kobject * kobj,
                          struct bin_attribute * attr);
int sysfs_remove_bin_file(struct kobject * kobj,
                          struct bin_attribute * attr);
```

Table 14: Binary Attributes

`udev` was written in 2003 to provide a dynamic device naming service based on user/administrator/distro-specified rules. It interacts with the `/sbin/hotplug` program, which gets called by the kernel when a variety of different events occur. udev uses information stored in sysfs about devices to name and configure them. More importantly, it is used as a building block by other components to provide a feature-rich and user-friendly device management environment.

Information about udev can be found at kernel.org [2]. Information about **HAL**—an aptly named hardware abstraction layer—can be found at freedesktop.org [1].

udev is based on **libsysfs**, a C library written to provide a robust programming interface for accessing sysfs objects and attributes. Information about libsysfs can be found at SourceForge [3]. The udev source contains a version of libsysfs that it builds against. On some distributions, it is already installed. If so, header files can be found in `/usr/include/sysfs/` and shared libraries can be found in `/lib/libsysfs.so.1`.

The **pciutils** package has been updated to use sysfs to access PCI configuration information, instead of using `/proc/bus/pci/`.

A simple application for extracting and parsing data from sysfs called **si** has been written. This utility can be used to display or modify any attribute, though its true benefit is efforts to aggregate and format subsystem-specific information into an intuitive format. At the time of publication, it is still in early alpha stages. It can be found at kernel.org [5].

12 Conclusion

sysfs is a filesystem that allows kernel subsystems to export kernel objects, object attributes, and object relationships to userspace. This information is strictly organized and usually formatted simply in ASCII, making it very accessible to users and applications. It provides a clear window into the kernel data structures and the physical or virtual objects that they control.

sysfs provides a core piece of infrastructure in the much larger effort of building flexible device and system management tools. To do this effectively, it retains a simple feature set that eases the use of its interfaces and data representations easy.

This paper has described the sysfs kernel interfaces and the userspace representation of kernel constructs. This paper has hopefully demystified sysfs enough to help readers understand what sysfs does and how it works, and with a bit of luck, encouraged them to dive head first into sysfs, whether it's from a developer standpoint, a user standpoint, or both.

References

[1] freedesktop.org. hal, 2005. `http://hal.freedesktop.org/wiki/Software_2fhal`.

[2] Greg Kroah-Hartman. udev, 2004. `http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html`.

[3] The libsysfs Developers. libsysfs, 2003. `http://linux-diag.sourceforge.net/Sysfsutils.html`.

[4] LWN.net. 2.6 driver porting series, 2003. `http://lwn.net/Articles/driver-porting/`.

[5] Patrick Mochel. si, 2005. `http://kernel.org/pub/linux/kernel/people/mochel/tools/si/`.

Proceedings of the Linux Symposium

Volume One

July 20nd–23th, 2005
Ottawa, Ontario
Canada