sysfs - _The_ filesystem for exporting kernel objects

Patrick Mochel <mochel@osdl.org>

Mike Murphy <mamurph@cs.clemson.edu>

Revised: 16 August 2011
Original: 10 January 2003

What it is:

sysfs is a ram-based filesystem initially based on ramfs. It provides a means to export kernel data structures, their attributes, and the linkages between them to userspace.

sysfs is tied inherently to the kobject infrastructure. Please read Documentation/core-api/kobject.rst for more information concerning the kobject interface.

Using sysfs

sysfs is always compiled in if CONFIG_SYSFS is defined. You can access it by doing:

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

Directory Creation

For every kobject that is registered with the system, a directory is created for it in sysfs. That directory is created as a subdirectory of the kobject's parent, expressing internal object hierarchies to userspace. Top-level directories in sysfs represent the common ancestors of object hierarchies; i.e. the subsystems the objects belong to.

Sysfs internally stores a pointer to the kobject that implements a directory in the kernfs_node object associated with the directory. In the past this kobject pointer has been used by sysfs to do reference counting directly on the kobject whenever the file is opened or closed. With the current sysfs implementation the kobject reference count is only modified directly by the function sysfs_schedule_callback().

Attributes

Attributes can be exported for kobjects in the form of regular files in the filesystem. Sysfs forwards file I/O operations to methods defined for the attributes, providing a means to read and write kernel attributes.

Attributes should be ASCII text files, preferably with only one value per file. It is noted that it may not be efficient to contain only one value per file, so it is socially acceptable to express an array of values of the same type.

Mixing types, expressing multiple lines of data, and doing fancy formatting of data is heavily frowned upon. Doing these things may get you publicly humiliated and your code rewritten without notice.

An attribute definition is simply:

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


int sysfs_create_file(struct kobject * kobj, const struct attribute * attr);
void sysfs_remove_file(struct kobject * kobj, const struct attribute * attr);
```

A bare attribute contains no means to read or write the value of the attribute. Subsystems are encouraged to define their own attribute structure and wrapper functions for adding and removing attributes for a specific object type.

For example, the driver model defines struct device_attribute like:

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

int device_create_file(struct device *, const struct device_attribute *);
void device_remove_file(struct device *, const struct device_attribute *);
```

It also defines this helper for defining device attributes:

```
#define DEVICE_ATTR(_name, _mode, _show, _store) \
struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _store)
```

For example, declaring:

```
static DEVICE_ATTR(foo, S_IWUSR | S_IRUGO, show_foo, store_foo);
```

is equivalent to doing:

```
static struct device_attribute dev_attr_foo = {
        .attr = {
                .name = "foo",
                .mode = S_IWUSR | S_IRUGO,
        },
        .show = show_foo,
        .store = store_foo,
};
```

Note as stated in include/linux/kernel.h "OTHER_WRITABLE? Generally considered a bad idea." so trying to set a sysfs file writable for everyone will fail reverting to RO mode for "Others".

For the common cases sysfs.h provides convenience macros to make defining attributes easier as well as making code more concise and readable. The above case could be shortened to:

static struct device_attribute dev_attr_foo = __ATTR_RW(foo);

the list of helpers available to define your wrapper function is:

__ATTR_RO(name):
: assumes default name_show and mode 0444

__ATTR_WO(name):
: assumes a name_store only and is restricted to mode 0200 that is root write access only.

__ATTR_RO_MODE(name, mode):
: fore more restrictive RO access currently only use case is the EFI System Resource Table (see drivers/firmware/efi/esrt.c)

__ATTR_RW(name):
: assumes default name_show, name_store and setting mode to 0644.

__ATTR_NULL:
: which sets the name to NULL and is used as end of list indicator (see: kernel/workqueue.c)

Subsystem-Specific Callbacks

When a subsystem defines a new attribute type, it must implement a set of sysfs operations for forwarding read and write calls to the show and store methods of the attribute owners:

```
struct sysfs_ops {
        ssize_t (*show)(struct kobject *, struct attribute *, char *);
        ssize_t (*store)(struct kobject *, struct attribute *, const char *, size_t);
};
```

[Subsystems should have already defined a struct kobj_type as a descriptor for this type, which is where the sysfs_ops pointer is stored. See the kobject documentation for more information.]

When a file is read or written, sysfs calls the appropriate method for the type. The method then translates the generic struct kobject and struct attribute pointers to the appropriate pointer types, and calls the associated methods.

To illustrate:

```
#define to_dev_attr(_attr) container_of(_attr, struct device_attribute, attr)

static ssize_t dev_attr_show(struct kobject *kobj, struct attribute *attr,
                             char *buf)
{
        struct device_attribute *dev_attr = to_dev_attr(attr);
        struct device *dev = kobj_to_dev(kobj);
        ssize_t ret = -EIO;

        if (dev_attr->show)
                ret = dev_attr->show(dev, dev_attr, buf);
        if (ret >= (ssize_t)PAGE_SIZE) {
                printk("dev_attr_show: %pS returned bad count\n",
                                dev_attr->show);
        }
        return ret;
}
```

Reading/Writing Attribute Data

To read or write attributes, show() or store() methods must be specified when declaring the attribute. The method types should be as simple as those defined for device attributes:

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

IOW, they should take only an object, an attribute, and a buffer as parameters.

sysfs allocates a buffer of size (PAGE_SIZE) and passes it to the method. Sysfs will call the method exactly once for each read or write. This forces the following behavior on the method implementations:

- On read(2), the show() method should fill the entire buffer. Recall that an attribute should only be exporting one value, or an array of similar values, so this shouldn't be that expensive.

  This allows userspace to do partial reads and forward seeks arbitrarily over the entire file at will. If userspace seeks back to zero or does a pread(2) with an offset of '0' the show() method will be called again, rearmed, to fill the buffer.

- On write(2), sysfs expects the entire buffer to be passed during the first write. Sysfs then passes the entire buffer to the store() method. A terminating null is added after the data on stores. This makes functions like sysfs_streq() safe to use.

  When writing sysfs files, userspace processes should first read the entire file, modify the values it wishes to change, then write the entire buffer back.

  Attribute method implementations should operate on an identical buffer when reading and writing values.

Other notes:

- Writing causes the show() method to be rearmed regardless of current file position.
- The buffer will always be PAGE_SIZE bytes in length. On i386, this is 4096.
- show() methods should return the number of bytes printed into the buffer.
- show() should only use sysfs_emit() or sysfs_emit_at() when formatting the value to be returned to user space.
- store() should return the number of bytes used from the buffer. If the entire buffer has been used, just return the count argument.
- show() or store() can always return errors. If a bad value comes through, be sure to return an error.
- The object passed to the methods will be pinned in memory via sysfs referencing counting its embedded object. However, the physical entity (e.g. device) the object represents may not be present. Be sure to have a way to check this, if necessary.

A very simple (and naive) implementation of a device attribute is:

```
static ssize_t show_name(struct device *dev, struct device_attribute *attr,
                         char *buf)
{
        return scnprintf(buf, PAGE_SIZE, "%s\n", dev->name);
}

static ssize_t store_name(struct device *dev, struct device_attribute *attr,
                          const char *buf, size_t count)
{
        snprintf(dev->name, sizeof(dev->name), "%.*s",
                 (int)min(count, sizeof(dev->name) - 1), buf);
        return count;
}

static DEVICE_ATTR(name, S_IRUGO, show_name, store_name);
```

(Note that the real implementation doesn't allow userspace to set the name for a device.)

Top Level Directory Layout

The sysfs directory arrangement exposes the relationship of kernel data structures.

The top level sysfs directory looks like:

```
block/
bus/
class/
dev/
devices/
firmware/
net/
fs/
```

devices/ contains a filesystem representation of the device tree. It maps directly to the internal kernel device tree, which is a hierarchy of struct device.

bus/ contains flat directory layout of the various bus types in the kernel. Each bus's directory contains two subdirectories:

```
devices/
```

```
drivers/
```

devices/ contains symlinks for each device discovered in the system that point to the device's directory under root/.

drivers/ contains a directory for each device driver that is loaded for devices on that particular bus (this assumes that drivers do not span multiple bus types).

fs/ contains a directory for some filesystems. Currently each filesystem wanting to export attributes must create its own hierarchy below fs/ (see ./fuse.txt for an example).

dev/ contains two directories char/ and block/. Inside these two directories there are symlinks named <major>:<minor>. These symlinks point to the sysfs directory for the given device. /sys/dev provides a quick way to lookup the sysfs interface for a device from the result of a stat(2) operation.

More information can driver-model specific features can be found in Documentation/driver-api/driver-model/.

TODO: Finish this section.

Current Interfaces

The following interface layers currently exist in sysfs:

devices (include/linux/device.h)

Structure:

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

Declaring:

```
DEVICE_ATTR(_name, _mode, _show, _store);
```

Creation/Removal:

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

bus drivers (include/linux/device.h)

Structure:

```
struct bus_attribute {
        struct attribute        attr;
        ssize_t (*show)(struct bus_type *, char * buf);
        ssize_t (*store)(struct bus_type *, const char * buf, size_t count);
};
```

Declaring:

```
static BUS_ATTR_RW(name);
static BUS_ATTR_RO(name);
static BUS_ATTR_WO(name);
```

Creation/Removal:

```
int bus_create_file(struct bus_type *, struct bus_attribute *);
void bus_remove_file(struct bus_type *, struct bus_attribute *);
```

device drivers (include/linux/device.h)

Structure:

```
struct driver_attribute {
        struct attribute        attr;
        ssize_t (*show)(struct device_driver *, char * buf);
        ssize_t (*store)(struct device_driver *, const char * buf,
                        size_t count);
};
```

Declaring:

```
DRIVER_ATTR_RO(_name)
DRIVER_ATTR_RW(_name)
```

Creation/Removal:

```
int driver_create_file(struct device_driver *, const struct driver_attribute *);
void driver_remove_file(struct device_driver *, const struct driver_attribute *);
```

Documentation

The sysfs directory structure and the attributes in each directory define an ABI between the kernel and user space. As for any ABI, it is important that this ABI is stable and properly documented. All new sysfs attributes must be documented in Documentation/ABI. See also Documentation/ABI/README for more information.