Configfs - Userspace-driven Kernel Object Configuration

Joel Becker <joel.becker@oracle.com>

Updated: 31 March 2005

What is configfs?

configfs is a ram-based filesystem that provides the converse of sysfs's functionality. Where sysfs is a filesystem-based view of kernel objects, configfs is a filesystem-based manager of kernel objects, or config_items.

With sysfs, an object is created in kernel (for example, when a device is discovered) and it is registered with sysfs. Its attributes then appear in sysfs, allowing userspace to read the attributes via readdir(3)/read(2). It may allow some attributes to be modified via write(2). The important point is that the object is created and destroyed in kernel, the kernel controls the lifecycle of the sysfs representation, and sysfs is merely a window on all this.

A configfs config_item is created via an explicit userspace operation: mkdir(2). It is destroyed via rmdir(2). The attributes appear at mkdir(2) time, and can be read or modified via read(2) and write(2). As with sysfs, readdir(3) queries the list of items and/or attributes. symlink(2) can be used to group items together. Unlike sysfs, the lifetime of the representation is completely driven by userspace. The kernel modules backing the items must respond to this.

Both sysfs and configfs can and should exist together on the same system. One is not a replacement for the other.

Using configfs

configfs can be compiled as a module or into the kernel. You can access it by doing:

```
mount -t configfs none /config
```

The configfs tree will be empty unless client modules are also loaded. These are modules that register their item types with configfs as subsystems. Once a client subsystem is loaded, it will appear as a subdirectory (or more than one) under /config. Like sysfs, the configfs tree is always there, whether mounted on /config or not.

An item is created via mkdir(2). The item's attributes will also appear at this time. readdir(3) can determine what the attributes are, read(2) can query their default values, and write(2) can store new values. Don't mix more than one attribute in one attribute file.

There are two types of configfs attributes:

- Normal attributes, which similar to sysfs attributes, are small ASCII text files, with a maximum size of one page (PAGE_SIZE, 4096 on i386). Preferably only one value per file should be used, and the same caveats from sysfs apply. Configfs expects write(2) to store the entire buffer at once. When writing to normal configfs attributes, userspace processes should first read the entire file, modify the portions they wish to change, and then write the entire buffer back.
- Binary attributes, which are somewhat similar to sysfs binary attributes, but with a few slight changes to semantics. The PAGE_SIZE limitation does not apply, but the whole binary item must fit in single kernel vmalloc'ed buffer. The write(2) calls from user space are buffered, and the attributes' write_bin_attribute method will be invoked on the final close, therefore it is imperative for user-space to check the return code of close(2) in order to verify that the operation finished successfully. To avoid a malicious user OOMing the kernel, there's a per-binary attribute maximum buffer value.

When an item needs to be destroyed, remove it with rmdir(2). An item cannot be destroyed if any other item has a link to it (via symlink(2)). Links can be removed via unlink(2).

Configuring FakeNBD: an Example

Imagine there's a Network Block Device (NBD) driver that allows you to access remote block devices. Call it FakeNBD. FakeNBD uses configfs for its configuration. Obviously, there will be a nice program that sysadmins use to configure FakeNBD, but somehow that program has to tell the driver about it. Here's where configfs comes in.

When the FakeNBD driver is loaded, it registers itself with configfs. readdir(3) sees this just fine:

```
# ls /config
fakenbd
```

A fakenbd connection can be created with mkdir(2). The name is arbitrary, but likely the tool will make some use of the name. Perhaps it is a uuid or a disk name:

```
# mkdir /config/fakenbd/disk1
# ls /config/fakenbd/disk1
target device rw
```

The target attribute contains the IP address of the server FakeNBD will connect to. The device attribute is the device on the server. Predictably, the rw attribute determines whether the connection is read-only or read-write:

```
# echo 10.0.0.1 > /config/fakenbd/disk1/target
# echo /dev/sda1 > /config/fakenbd/disk1/device
# echo 1 > /config/fakenbd/disk1/rw
```

That's it. That's all there is. Now the device is configured, via the shell no less.

Coding With configfs

Every object in configfs is a config_item. A config_item reflects an object in the subsystem. It has attributes that match values on that object. configfs handles the filesystem representation of that object and its attributes, allowing the subsystem to ignore all but the basic show/store interaction.

Items are created and destroyed inside a config_group. A group is a collection of items that share the same attributes and operations. Items are created by mkdir(2) and removed by rmdir(2), but configfs handles that. The group has a set of operations to perform these tasks

A subsystem is the top level of a client module. During initialization, the client module registers the subsystem with configfs, the subsystem appears as a directory at the top of the configfs filesystem. A subsystem is also a config_group, and can do everything a config_group can.

struct config_item

```
struct config_item {
        char                    *ci_name;
        char                    ci_namebuf[UOBJ_NAME_LEN];
        struct kref             ci_kref;
        struct list_head        ci_entry;
        struct config_item      *ci_parent;
        struct config_group     *ci_group;
        struct config_item_type *ci_type;
        struct dentry           *ci_dentry;
};

void config_item_init(struct config_item *);
void config_item_init_type_name(struct config_item *,
                                const char *name,
                                struct config_item_type *type);
struct config_item *config_item_get(struct config_item *);
void config_item_put(struct config_item *);
```

Generally, struct config_item is embedded in a container structure, a structure that actually represents what the subsystem is doing. The config_item portion of that structure is how the object interacts with configfs.

Whether statically defined in a source file or created by a parent config_group, a config_item must have one of the _init() functions called on it. This initializes the reference count and sets up the appropriate fields.

All users of a config_item should have a reference on it via config_item_get(), and drop the reference when they are done via config_item_put().

By itself, a config_item cannot do much more than appear in configfs. Usually a subsystem wants the item to display and/or store attributes, among other things. For that, it needs a type.

struct config_item_type

```
struct configfs_item_operations {
        void (*release)(struct config_item *);
        int (*allow_link)(struct config_item *src,
                          struct config_item *target);
        void (*drop_link)(struct config_item *src,
                         struct config_item *target);
};

struct config_item_type {
        struct module                           *ct_owner;
        struct configfs_item_operations         *ct_item_ops;
        struct configfs_group_operations        *ct_group_ops;
        struct configfs_attribute               **ct_attrs;
        struct configfs_bin_attribute           **ct_bin_attrs;
};
```

The most basic function of a config_item_type is to define what operations can be performed on a config_item. All items that have been allocated dynamically will need to provide the ct_item_ops->release() method. This method is called when the config_item's reference count reaches zero.

struct configfs_attribute

```
struct configfs_attribute {
        char                    *ca_name;
        struct module           *ca_owner;
        umode_t                  ca_mode;
        ssize_t (*show)(struct config_item *, char *);
        ssize_t (*store)(struct config_item *, const char *, size_t);
};
```

When a config_item wants an attribute to appear as a file in the item's configfs directory, it must define a configfs_attribute describing it. It then adds the attribute to the NULL-terminated array config_item_type->ct_attrs. When the item appears in configfs, the attribute file will appear with the configfs_attribute->ca_name filename. configfs_attribute->ca_mode specifies the file permissions.

If an attribute is readable and provides a ->show method, that method will be called whenever userspace asks for a read(2) on the attribute. If an attribute is writable and provides a ->store method, that method will be called whenever userspace asks for a write(2) on the attribute.

struct configfs_bin_attribute

```
struct configfs_bin_attribute {
        struct configfs_attribute       cb_attr;
        void                            *cb_private;
        size_t                          cb_max_size;
};
```

The binary attribute is used when the one needs to use binary blob to appear as the contents of a file in the item's configfs directory. To do so add the binary attribute to the NULL-terminated array config_item_type->ct_bin_attrs, and the item appears in configfs, the attribute file will appear with the configfs_bin_attribute->cb_attr.ca_name filename. configfs_bin_attribute->cb_attr.ca_mode specifies the file permissions. The cb_private member is provided for use by the driver, while the cb_max_size member specifies the maximum amount of vmalloc buffer to be used.

If binary attribute is readable and the config_item provides a ct_item_ops->read_bin_attribute() method, that method will be called whenever userspace asks for a read(2) on the attribute. The converse will happen for write(2). The reads/writes are bufferred so only a single read/write will occur; the attributes' need not concern itself with it.

struct config_group

A config_item cannot live in a vacuum. The only way one can be created is via mkdir(2) on a config_group. This will trigger creation of a child item:

```
struct config_group {
        struct config_item              cg_item;
        struct list_head                cg_children;
        struct configfs_subsystem       *cg_subsys;
        struct list_head                default_groups;
        struct list_head                group_entry;
};

void config_group_init(struct config_group *group);
void config_group_init_type_name(struct config_group *group,
                                 const char *name,
                                 struct config_item_type *type);
```

The config_group structure contains a config_item. Properly configuring that item means that a group can behave as an item in its own right. However, it can do more: it can create child items or groups. This is accomplished via the group operations specified on the group's config_item_type:

```
struct configfs_group_operations {
        struct config_item *(*make_item)(struct config_group *group,
                                         const char *name);
        struct config_group *(*make_group)(struct config_group *group,
                                           const char *name);
        int (*commit_item)(struct config_item *item);
        void (*disconnect_notify)(struct config_group *group,
                                  struct config_item *item);
        void (*drop_item)(struct config_group *group,
                          struct config_item *item);
};
```

A group creates child items by providing the ct_group_ops->make_item() method. If provided, this method is called from mkdir(2) in the group's directory. The subsystem allocates a new config_item (or more likely, its container structure), initializes it, and returns it to configfs. Configfs will then populate the filesystem tree to reflect the new item.

If the subsystem wants the child to be a group itself, the subsystem provides ct_group_ops->make_group(). Everything else behaves the same, using the group _init() functions on the group.

Finally, when userspace calls rmdir(2) on the item or group, ct_group_ops->drop_item() is called. As a config_group is also a config_item, it is not necessary for a separate drop_group() method. The subsystem must config_item_put() the reference that was initialized upon item allocation. If a subsystem has no work to do, it may omit the ct_group_ops->drop_item() method, and configfs will call config_item_put() on the item on behalf of the subsystem.

Important:

> drop_item() is void, and as such cannot fail. When rmdir(2) is called, configfs WILL remove the item from the filesystem tree (assuming that it has no children to keep it busy). The subsystem is responsible for responding to this. If the subsystem has references to the item in other threads, the memory is safe. It may take some time for the item to actually disappear from the subsystem's usage. But it is gone from configfs.

When drop_item() is called, the item's linkage has already been torn down. It no longer has a reference on its parent and has no place in the item hierarchy. If a client needs to do some cleanup before this teardown happens, the subsystem can implement the ct_group_ops->disconnect_notify() method. The method is called after configfs has removed the item from the filesystem view but before the item is removed from its parent group. Like drop_item(), disconnect_notify() is void and cannot fail. Client subsystems should not drop any references here, as they still must do it in drop_item().

A config_group cannot be removed while it still has child items. This is implemented in the configfs rmdir(2) code. ->drop_item() will not be called, as the item has not been dropped. rmdir(2) will fail, as the directory is not empty.

struct configfs_subsystem

A subsystem must register itself, usually at module_init time. This tells configfs to make the subsystem appear in the file tree:

```
struct configfs_subsystem {
        struct config_group     su_group;
        struct mutex            su_mutex;
};

int configfs_register_subsystem(struct configfs_subsystem *subsys);
void configfs_unregister_subsystem(struct configfs_subsystem *subsys);
```

A subsystem consists of a toplevel config_group and a mutex. The group is where child config_items are created. For a subsystem, this group is usually defined statically. Before calling configfs_register_subsystem(), the subsystem must have initialized the group via the usual group _init() functions, and it must also have initialized the mutex.

When the register call returns, the subsystem is live, and it will be visible via configfs. At that point, mkdir(2) can be called and the subsystem must be ready for it.

An Example

The best example of these basic concepts is the simple_children subsystem/group and the simple_child item in samples/configfs/configfs_sample.c. It shows a trivial object displaying and storing an attribute, and a simple group creating and destroying these children.

Hierarchy Navigation and the Subsystem Mutex

There is an extra bonus that configfs provides. The config_groups and config_items are arranged in a hierarchy due to the fact that they appear in a filesystem. A subsystem is NEVER to touch the filesystem parts, but the subsystem might be interested in this hierarchy. For this reason, the hierarchy is mirrored via the config_group->cg_children and config_item->ci_parent structure members.

A subsystem can navigate the cg_children list and the ci_parent pointer to see the tree created by the subsystem. This can race with configfs' management of the hierarchy, so configfs uses the subsystem mutex to protect modifications. Whenever a subsystem wants to navigate the hierarchy, it must do so under the protection of the subsystem mutex.

A subsystem will be prevented from acquiring the mutex while a newly allocated item has not been linked into this hierarchy. Similarly, it will not be able to acquire the mutex while a dropping item has not yet been unlinked. This means that an item's ci_parent pointer will never be NULL while the item is in configfs, and that an item will only be in its parent's cg_children list for the same duration. This allows a subsystem to trust ci_parent and cg_children while they hold the mutex.

Item Aggregation Via symlink(2)

configfs provides a simple group via the group->item parent/child relationship. Often, however, a larger environment requires aggregation outside of the parent/child connection. This is implemented via symlink(2).

A config_item may provide the ct_item_ops->allow_link() and ct_item_ops->drop_link() methods. If the ->allow_link() method exists, symlink(2) may be called with the config_item as the source of the link. These links are only allowed between configfs config_items. Any symlink(2) attempt outside the configfs filesystem will be denied.

When symlink(2) is called, the source config_item's ->allow_link() method is called with itself and a target item. If the source item allows linking to target item, it returns 0. A source item may wish to reject a link if it only wants links to a certain type of object (say, in its own subsystem).

When unlink(2) is called on the symbolic link, the source item is notified via the ->drop_link() method. Like the ->drop_item() method, this is a void function and cannot return failure. The subsystem is responsible for responding to the change.

A config_item cannot be removed while it links to any other item, nor can it be removed while an item links to it. Dangling symlinks are not allowed in configfs.

Automatically Created Subgroups

A new config_group may want to have two types of child config_items. While this could be codified by magic names in ->make_item(), it is much more explicit to have a method whereby userspace sees this divergence.

Rather than have a group where some items behave differently than others, configfs provides a method whereby one or many subgroups are automatically created inside the parent at its creation. Thus, mkdir("parent") results in "parent", "parent/subgroup1", up through "parent/subgroupN". Items of type 1 can now be created in "parent/subgroup1", and items of type N can be created in "parent/subgroupN".

These automatic subgroups, or default groups, do not preclude other children of the parent group. If ct_group_ops->make_group() exists, other child groups can be created on the parent group directly.

A configfs subsystem specifies default groups by adding them using the configfs_add_default_group() function to the parent config_group structure. Each added group is populated in the configfs tree at the same time as the parent group. Similarly, they are removed at the same time as the parent. No extra notification is provided. When a ->drop_item() method call notifies the subsystem the parent group is going away, it also means every default group child associated with that parent group.

As a consequence of this, default groups cannot be removed directly via rmdir(2). They also are not considered when rmdir(2) on the parent group is checking for children.

Dependent Subsystems

Sometimes other drivers depend on particular configfs items. For example, ocfs2 mounts depend on a heartbeat region item. If that region item is removed with rmdir(2), the ocfs2 mount must BUG or go readonly. Not happy.

configfs provides two additional API calls: configfs_depend_item() and configfs_undepend_item(). A client driver can call configfs_depend_item() on an existing item to tell configfs that it is depended on. configfs will then return -EBUSY from rmdir(2) for that item. When the item is no longer depended on, the client driver calls configfs_undepend_item() on it.

These API cannot be called underneath any configfs callbacks, as they will conflict. They can block and allocate. A client driver probably shouldn't calling them of its own gumption. Rather it should be providing an API that external subsystems call.

How does this work? Imagine the ocfs2 mount process. When it mounts, it asks for a heartbeat region item. This is done via a call into the heartbeat code. Inside the heartbeat code, the region item is looked up. Here, the heartbeat code calls configfs_depend_item(). If it succeeds, then heartbeat knows the region is safe to give to ocfs2. If it fails, it was being torn down anyway, and heartbeat can gracefully pass up an error.

Committable Items

Note:

Committable items are currently unimplemented.

Some config_items cannot have a valid initial state. That is, no default values can be specified for the item's attributes such that the item can do its work. Userspace must configure one or more attributes, after which the subsystem can start whatever entity this item represents.

Consider the FakeNBD device from above. Without a target address *and* a target device, the subsystem has no idea what block device to import. The simple example assumes that the subsystem merely waits until all the appropriate attributes are configured, and then connects. This will, indeed, work, but now every attribute store must check if the attributes are initialized. Every attribute store must fire off the connection if that condition is met.

Far better would be an explicit action notifying the subsystem that the config_item is ready to go. More importantly, an explicit action allows the subsystem to provide feedback as to whether the attributes are initialized in a way that makes sense. configfs provides this as committable items.

configfs still uses only normal filesystem operations. An item is committed via rename(2). The item is moved from a directory where it can be modified to a directory where it cannot.

Any group that provides the ct_group_ops->commit_item() method has committable items. When this group appears in configfs, mkdir(2) will not work directly in the group. Instead, the group will have two subdirectories: "live" and "pending". The "live" directory does not support mkdir(2) or rmdir(2) either. It only allows rename(2). The "pending" directory does allow mkdir(2) and rmdir(2). An item is created in the "pending" directory. Its attributes can be modified at will. Userspace commits the item by renaming it into the "live" directory. At this point, the subsystem receives the ->commit_item() callback. If all required attributes are filled to satisfaction, the method returns zero and the item is moved to the "live" directory.

As rmdir(2) does not work in the "live" directory, an item must be shutdown, or "uncommitted". Again, this is done via rename(2), this time from the "live" directory back to the "pending" one. The subsystem is notified by the ct_group_ops->uncommit_object() method.