

Everything you never wanted to know about kobjects, ksets, and ktypes

Author: Greg Kroah-Hartman <gregkh@linuxfoundation.org>
Last updated: December 19, 2007

Based on an original article by Jon Corbet for lwn.net written October 1, 2003 and located at <https://lwn.net/Articles/51437/>

Part of the difficulty in understanding the driver model - and the kobject abstraction upon which it is built - is that there is no obvious starting place. Dealing with kobjects requires understanding a few different types, all of which make reference to each other. In an attempt to make things easier, we'll take a multi-pass approach, starting with vague terms and adding detail as we go. To that end, here are some quick definitions of some terms we will be working with.

- A kobject is an object of type struct kobject. Kobjects have a name and a reference count. A kobject also has a parent pointer (allowing objects to be arranged into hierarchies), a specific type, and, usually, a representation in the sysfs virtual filesystem.

Kobjects are generally not interesting on their own; instead, they are usually embedded within some other structure which contains the stuff the code is really interested in.

No structure should **EVER** have more than one kobject embedded within it. If it does, the reference counting for the object is sure to be messed up and incorrect, and your code will be buggy. So do not do this.

- A ktype is the type of object that embeds a kobject. Every structure that embeds a kobject needs a corresponding ktype. The ktype controls what happens to the kobject when it is created and destroyed.
- A kset is a group of kobjects. These kobjects can be of the same ktype or belong to different ktypes. The kset is the basic container type for collections of kobjects. Ksets contain their own kobjects, but you can safely ignore that implementation detail as the kset core code handles this kobject automatically.

When you see a sysfs directory full of other directories, generally each of those directories corresponds to a kobject in the same kset.

We'll look at how to create and manipulate all of these types. A bottom-up approach will be taken, so we'll go back to kobjects.

Embedding kobjects

It is rare for kernel code to create a standalone kobject, with one major exception explained below. Instead, kobjects are used to control access to a larger, domain-specific object. To this end, kobjects will be found embedded in other structures. If you are used to thinking of things in object-oriented terms, kobjects can be seen as a top-level, abstract class from which other classes are derived. A kobject implements a set of capabilities which are not particularly useful by themselves, but are nice to have in other objects. The C language does not allow for the direct expression of inheritance, so other techniques - such as structure embedding - must be used.

(As an aside, for those familiar with the kernel linked list implementation, this is analogous as to how "list_head" structs are rarely useful on their own, but are invariably found embedded in the larger objects of interest.)

So, for example, the UIO code in `drivers/uio/uio.c` has a structure that defines the memory region associated with a uio device:

```
struct uio_map {
    struct kobject kobj;
    struct uio_mem *mem;
};
```

If you have a struct `uio_map` structure, finding its embedded kobject is just a matter of using the `kobj` member. Code that works with kobjects will often have the opposite problem, however: given a struct kobject pointer, what is the pointer to the containing structure? You must avoid tricks (such as assuming that the kobject is at the beginning of the structure) and, instead, use the `container_of()` macro, found in `<linux/kernel.h>`:

```
container_of(ptr, type, member)
```

where:

- `ptr` is the pointer to the embedded kobject,
- `type` is the type of the containing structure, and
- `member` is the name of the structure field to which pointer points.

The return value from `container_of()` is a pointer to the corresponding container type. So, for example, a pointer `kp` to a struct kobject embedded **within** a struct `uio_map` could be converted to a pointer to the **containing** `uio_map` structure with:

```
struct uio_map *u_map = container_of(kp, struct uio_map, kobj);
```

For convenience, programmers often define a simple macro for **back-casting** kobject pointers to the containing type. Exactly this happens in the earlier `drivers/uid/uid.c`, as you can see here:

```
struct uid_map {
    struct kobject kobj;
    struct uid_mem *mem;
};

#define to_map(map) container_of(map, struct uid_map, kobj)
```

where the macro argument "map" is a pointer to the struct kobject in question. That macro is subsequently invoked with:

```
struct uid_map *map = to_map(kobj);
```

Initialization of kobjects

Code which creates a kobject must, of course, initialize that object. Some of the internal fields are setup with a (mandatory) call to `kobject_init()`:

```
void kobject_init(struct kobject *kobj, const struct kobj_type *ktype);
```

The `ktype` is required for a kobject to be created properly, as every kobject must have an associated `kobj_type`. After calling `kobject_init()`, to register the kobject with sysfs, the function `kobject_add()` must be called:

```
int kobject_add(struct kobject *kobj, struct kobject *parent,
               const char *fmt, ...);
```

This sets up the parent of the kobject and the name for the kobject properly. If the kobject is to be associated with a specific kset, `kobj->kset` must be assigned before calling `kobject_add()`. If a kset is associated with a kobject, then the parent for the kobject can be set to NULL in the call to `kobject_add()` and then the kobject's parent will be the kset itself.

As the name of the kobject is set when it is added to the kernel, the name of the kobject should never be manipulated directly. If you must change the name of the kobject, call `kobject_rename()`:

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

`kobject_rename()` does not perform any locking or have a solid notion of what names are valid so the caller must provide their own sanity checking and serialization.

There is a function called `kobject_set_name()` but that is legacy cruft and is being removed. If your code needs to call this function, it is incorrect and needs to be fixed.

To properly access the name of the kobject, use the function `kobject_name()`:

```
const char *kobject_name(const struct kobject * kobj);
```

There is a helper function to both initialize and add the kobject to the kernel at the same time, called surprisingly enough `kobject_init_and_add()`:

```
int kobject_init_and_add(struct kobject *kobj, const struct kobj_type *ktype,
                        struct kobject *parent, const char *fmt, ...);
```

The arguments are the same as the individual `kobject_init()` and `kobject_add()` functions described above.

Uevents

After a kobject has been registered with the kobject core, you need to announce to the world that it has been created. This can be done with a call to `kobject_uevent()`:

```
int kobject_uevent(struct kobject *kobj, enum kobject_action action);
```

Use the **KOBJ_ADD** action for when the kobject is first added to the kernel. This should be done only after any attributes or children of the kobject have been initialized properly, as userspace will instantly start to look for them when this call happens.

When the kobject is removed from the kernel (details on how to do that are below), the uevent for **KOBJ_REMOVE** will be automatically created by the kobject core, so the caller does not have to worry about doing that by hand.

Reference counts

One of the key functions of a kobject is to serve as a reference counter for the object in which it is embedded. As long as references to the object exist, the object (and the code which supports it) must continue to exist. The low-level functions for manipulating a kobject's reference counts are:

```
struct kobject *kobject_get(struct kobject *kobj);
void kobject_put(struct kobject *kobj);
```

A successful call to `kobject_get()` will increment the kobject's reference counter and return the pointer to the kobject.

When a reference is released, the call to `kobject_put()` will decrement the reference count and, possibly, free the object. Note that `kobject_init()` sets the reference count to one, so the code which sets up the kobject will need to do a `kobject_put()` eventually to release that reference.

Because kobjects are dynamic, they must not be declared statically or on the stack, but instead, always allocated dynamically. Future versions of the kernel will contain a run-time check for kobjects that are created statically and will warn the developer of this improper usage.

If all that you want to use a kobject for is to provide a reference counter for your structure, please use the `struct kref` instead; a kobject would be overkill. For more information on how to use `struct kref`, please see the file `Documentation/core-api/kref.rst` in the Linux kernel source tree.

Creating "simple" kobjects

Sometimes all that a developer wants is a way to create a simple directory in the `sysfs` hierarchy, and not have to mess with the whole complication of `ksets`, `show` and `store` functions, and other details. This is the one exception where a single kobject should be created. To create such an entry, use the function:

```
struct kobject *kobject_create_and_add(const char *name, struct kobject *parent);
```

This function will create a kobject and place it in `sysfs` in the location underneath the specified parent kobject. To create simple attributes associated with this kobject, use:

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

or:

```
int sysfs_create_group(struct kobject *kobj, const struct attribute_group *grp);
```

Both types of attributes used here, with a kobject that has been created with the `kobject_create_and_add()`, can be of type `kobj_attribute`, so no special custom attribute is needed to be created.

See the example module, `samples/kobject/kobject-example.c` for an implementation of a simple kobject and attributes.

ktypes and release methods

One important thing still missing from the discussion is what happens to a kobject when its reference count reaches zero. The code which created the kobject generally does not know when that will happen; if it did, there would be little point in using a kobject in the first place. Even predictable object lifecycles become more complicated when `sysfs` is brought in as other portions of the kernel can get a reference on any kobject that is registered in the system.

The end result is that a structure protected by a kobject cannot be freed before its reference count goes to zero. The reference count is not under the direct control of the code which created the kobject. So that code must be notified asynchronously whenever the last reference to one of its kobjects goes away.

Once you registered your kobject via `kobject_add()`, you must never use `kfree()` to free it directly. The only safe way is to use `kobject_put()`. It is good practice to always use `kobject_put()` after `kobject_init()` to avoid errors creeping in.

This notification is done through a kobject's `release()` method. Usually such a method has a form like:

```
void my_object_release(struct kobject *kobj)
{
    struct my_object *mine = container_of(kobj, struct my_object, kobj);

    /* Perform any additional cleanup on this object, then... */
    kfree(mine);
}
```

One important point cannot be overstated: every kobject must have a `release()` method, and the kobject must persist (in a consistent state) until that method is called. If these constraints are not met, the code is flawed. Note that the kernel will warn you if you forget to provide a `release()` method. Do not try to get rid of this warning by providing an "empty" release function.

If all your cleanup function needs to do is call `kfree()`, then you must create a wrapper function which uses `container_of()` to upcast to the correct type (as shown in the example above) and then calls `kfree()` on the overall structure.

Note, the name of the kobject is available in the release function, but it must NOT be changed within this callback. Otherwise there will be a memory leak in the kobject core, which makes people unhappy.

Interestingly, the `release()` method is not stored in the kobject itself; instead, it is associated with the `ktype`. So let us introduce `struct kobj_type`:

```
struct kobj_type {
    void (*release)(struct kobject *kobj);
    const struct sysfs_ops *sysfs_ops;
    const struct attribute_group **default_groups;
```

```

const struct kobj_ns_type_operations *(*child_ns_type)(struct kobject *kobj);
const void *(*namespace)(struct kobject *kobj);
void (*get_ownership)(struct kobject *kobj, kuid_t *uid, kgid_t *gid);
};

```

This structure is used to describe a particular type of `kobject` (or, more correctly, of containing object). Every `kobject` needs to have an associated `kobj_type` structure; a pointer to that structure must be specified when you call `kobject_init()` or `kobject_init_and_add()`.

The `release` field in `struct kobj_type` is, of course, a pointer to the `release()` method for this type of `kobject`. The other two fields (`sysfs_ops` and `default_groups`) control how objects of this type are represented in `sysfs`; they are beyond the scope of this document.

The `default_groups` pointer is a list of default attributes that will be automatically created for any `kobject` that is registered with this `ktype`.

ksets

A `kset` is merely a collection of `kobjects` that want to be associated with each other. There is no restriction that they be of the same `ktype`, but be very careful if they are not.

A `kset` serves these functions:

- It serves as a bag containing a group of objects. A `kset` can be used by the kernel to track "all block devices" or "all PCI device drivers."
- A `kset` is also a subdirectory in `sysfs`, where the associated `kobjects` with the `kset` can show up. Every `kset` contains a `kobject` which can be set up to be the parent of other `kobjects`; the top-level directories of the `sysfs` hierarchy are constructed in this way.
- `Ksets` can support the "hotplugging" of `kobjects` and influence how `uevent` events are reported to user space.

In object-oriented terms, "`kset`" is the top-level container class; `ksets` contain their own `kobject`, but that `kobject` is managed by the `kset` code and should not be manipulated by any other user.

A `kset` keeps its children in a standard kernel linked list. `Kobjects` point back to their containing `kset` via their `kset` field. In almost all cases, the `kobjects` belonging to a `kset` have that `kset` (or, strictly, its embedded `kobject`) in their parent.

As a `kset` contains a `kobject` within it, it should always be dynamically created and never declared statically or on the stack. To create a new `kset` use:

```

struct kset *kset_create_and_add(const char *name,
                                const struct kset_uevent_ops *uevent_ops,
                                struct kobject *parent_kobj);

```

When you are finished with the `kset`, call:

```

void kset_unregister(struct kset *k);

```

to destroy it. This removes the `kset` from `sysfs` and decrements its reference count. When the reference count goes to zero, the `kset` will be released. Because other references to the `kset` may still exist, the release may happen after `kset_unregister()` returns.

An example of using a `kset` can be seen in the `samples/kobject/kset-example.c` file in the kernel tree.

If a `kset` wishes to control the `uevent` operations of the `kobjects` associated with it, it can use the `struct kset_uevent_ops` to handle it:

```

struct kset_uevent_ops {
    int (* const filter)(struct kobject *kobj);
    const char *(* const name)(struct kobject *kobj);
    int (* const uevent)(struct kobject *kobj, struct kobj_uevent_env *env);
};

```

The `filter` function allows a `kset` to prevent a `uevent` from being emitted to userspace for a specific `kobject`. If the function returns 0, the `uevent` will not be emitted.

The `name` function will be called to override the default name of the `kset` that the `uevent` sends to userspace. By default, the name will be the same as the `kset` itself, but this function, if present, can override that name.

The `uevent` function will be called when the `uevent` is about to be sent to userspace to allow more environment variables to be added to the `uevent`.

One might ask how, exactly, a `kobject` is added to a `kset`, given that no functions which perform that function have been presented. The answer is that this task is handled by `kobject_add()`. When a `kobject` is passed to `kobject_add()`, its `kset` member should point to the `kset` to which the `kobject` will belong. `kobject_add()` will handle the rest.

If the `kobject` belonging to a `kset` has no parent `kobject` set, it will be added to the `kset`'s directory. Not all members of a `kset` do necessarily live in the `kset` directory. If an explicit parent `kobject` is assigned before the `kobject` is added, the `kobject` is registered with the `kset`, but added below the parent `kobject`.

Kobject removal

After a kobject has been registered with the kobject core successfully, it must be cleaned up when the code is finished with it. To do that, call `kobject_put()`. By doing this, the kobject core will automatically clean up all of the memory allocated by this kobject. If a `KOBJ_ADD` uevent has been sent for the object, a corresponding `KOBJ_REMOVE` uevent will be sent, and any other sysfs housekeeping will be handled for the caller properly.

If you need to do a two-stage delete of the kobject (say you are not allowed to sleep when you need to destroy the object), then call `kobject_del()` which will unregister the kobject from sysfs. This makes the kobject "invisible", but it is not cleaned up, and the reference count of the object is still the same. At a later time call `kobject_put()` to finish the cleanup of the memory associated with the kobject.

`kobject_del()` can be used to drop the reference to the parent object, if circular references are constructed. It is valid in some cases, that a parent object's references a child. Circular references must be broken with an explicit call to `kobject_del()`, so that a release function will be called, and the objects in the former circle release each other.

Example code to copy from

For a more complete example of using ksets and kobjects properly, see the example programs `samples/kobject/{kobject-example.c,kset-example.c}`, which will be built as loadable modules if you select `CONFIG_SAMPLE_KOBJECT`.