kobjects and krefs

lockless reference counting for kernel structures

Greg Kroah-Hartman [*]
Linux Technology Center
IBM Corp.
greg@kroah.com
gregkh@us.ibm.com

Abstract

This paper will describe the current kobject and kref kernel structures in detail. It will cover why they were created, how to use them, and how the internals work. It will also cover a few directions that these structures might be taking in the future.

1 Introduction

The Linux kernel file `Documentation/CodingStyle` has the following statement about reference counting:

> Data structures that have visibility outside the single-threaded environment they are created and destroyed in should always have reference counts. In the kernel, garbage collection doesn't exist (and outside the kernel garbage collection is slow and inefficient), which means that you absolutely _have_ to reference count all your uses.

This requirement of providing proper reference counting for kernel structures has caused developers to create their own logic and functions to implement this feature. During the development of the Linux Kernel Driver model[4], a simple structure, `struct kobject`, was created that provided automatic reference counting for any user of the object. Unfortunately, `struct kobject` is closely tied to the kernel driver model, and for any data structure that does not want to show up in sysfs, and participate in the global kernel "web woven by a spider on drugs"[2], using a struct kobject only for reference counting is a big waste of memory resources and is much more complex than needed. To this end, the data structure, `struct kref`, was created to provide a simple, and hopefully failproof method of adding proper reference counting to any kernel data structure.

2 How to use it

To use the `struct kref` structure, simply embed it within the structure that reference counting is needed for. For example, to add reference counting to a structure called `struct foo` then it would be defined as:

```
struct foo {
    ...
    struct kref kref;
    ...
```

[*]This work represents the view of the author and does not necessarily represent the view of IBM.

```
};
```

It is not important that the `struct kref` structure be the first or last element of the structure that it is embedded in. The only requirement is that the whole `struct kref` structure be in the structure being reference counted, not a pointer to the a `struct kref` structure.

When the `struct foo` structure is initialized, the `kref` variable must also be initialized before reference counting can be used. This is done with a call to the `kref_init` function:

```
struct foo *foo;
foo = kmalloc(sizeof(*foo),
              GFP_KERNEL);
kref_init(&foo->kref,
          foo_release);
```

The parameter `foo_release` is a pointer The first parameter of `kref_init` is a pointer to the `struct kref` structure that is to be initialized. The second parameter is a pointer to the release function for the structure. This release function is described in detail below.

After the kref structure has been initialized, the internal reference count of the structure is set to 1. Now the reference count can be incremented and decremented at will.

To increment the reference count of a kref structure, the function `kref_get` is called:

```
/* get a new reference to our
   foo structure */
kref_get(&foo->kref);
```

When a user of the structure is finished with it, the `kref_put` function should be called to release the reference:

```
/* finished with this
   foo structure */
kref_put(&foo->kref);
```

This function should also be called after the original creator of the structure that the kref variable is in, is finished with the structure. The `kfree` function must *NOT* be directly called because other portions of the kernel could have valid references to this structure.

After the `kref_put` function is called, the structure can not be referred to by any future code, as the memory for that structure could be now gone.

When the last reference count is released, the function that was passed to the original `kref_init` function is called to release the memory used by the structure. The prototype of this function must accept a pointer to a `struct kref`:

```
void foo_release(struct kref
                 *kref)
{
    struct foo *foo;

    foo = container_of(foo,
                       struct foo,
                       kref);
    kfree(foo);
}
```

As the above example function shows, to get back to the original `struct foo` structure location, the `container_of` macro is used. For a complete description of how the `container_of` macro works, please see[1].

As there are not any locks within the `kref` structure, there are three rules that need to be followed when using this reference counting logic:

- If the code accessing the variable already has a valid reference to the structure, it is safe, and required to increment that reference with a call to `kref_get` in order to give the variable to any other piece of code.

- If the code accessing the variable already has a valid reference to the structure, then it is safe to release that reference with a call to `kref_put`.
- If the code wanting to access the variable, does not have a valid reference, then it needs to serialize with a place within the code where the last call to `kref_put` put could happen.

This last rule can not be emphasized enough. The only reason that the `struct kref` can work without any internal locks is because a call to `kref_get` can not happen at the same time that `kref_put` is happening. In order to ensure this, a simple lock for the driver or subsystem that owns the specific `struct kref` reference can be used.

An example of using such a lock can be seen in Figure 1.

So, with the three simple functions, `kref_init`, `kref_get`, and `kref_put`, combined with a release function that the caller provides, complete reference counting can be added to any kernel structure.

3 How it works

`struct kref` is a very tiny structure with only two elements:

```
struct kref {
    atomic_t refcount;
    void (*release)(struct kref *kref);
};
```

The `refcount` variable is an atomic counter that is used to hold the reference count of the structure. The `release` variable is a pointer to a function that will be called when the last user of the structure is finished with the structure.

The `kref_init` function is a mere three lines long:

```
void kref_init(struct kref *kref,
               void (*release)
               (struct kref *kref))
{
    WARN_ON(release == NULL);
    atomic_set(&kref->refcount,1);
    kref->release = release;
}
```

First a warning is printed out to the syslog if a `release` callback is not provided, as this is not allowed. Then the `refcount` variable is initialized to 1 as the structure needs to have a single initial reference count. After that the `release` function pointer is stored in the `release` variable in the structure.

The `kref_get` function is also only three lines of code:

```
struct kref *kref_get(struct kref *kref)
{
    WARN_ON(!atomic_read(&kref->refcount));
    atomic_inc(&kref->refcount);
    return kref;
}
```

Again, a warning is printed out to the syslog if the `refcount` variable is zero. This catches the very common error of calling `kref_get` without first calling `kref_init`. After that, the `refcount` variable is incremented, and then a pointer to the same structure is returned. This return type makes it easier for code to do things pass the result of `kref_get` as a function parameter:

```
do_foo(kref_get(my_kref));
```

Keeping with the tradition of tiny functions, the `kref_put` function weighs in at a whopping two lines:

```
/* prevent races between open() and disconnect() */
static DECLARE_MUTEX (disconnect_sem);

static int skel_open(struct inode *inode, struct file *file)
{
    struct usb_skel *dev;
    struct usb_interface *interface;

    /* prevent disconnects */
    down (&disconnect_sem);

    interface = usb_find_interface(&skel_driver, iminor(inode));
    dev = usb_get_intfdata(interface);

    /* increment our usage count for the device */
    kref_get(&dev->kref);
    up(&disconnect_sem);

    ...
}

static void skel_disconnect(struct usb_interface *interface)
{
    struct usb_skel *dev;
    int minor = interface->minor;

    /* prevent skel_open() from racing skel_disconnect() */
    down (&disconnect_sem);

    dev = usb_get_intfdata(interface);
    usb_set_intfdata(interface, NULL);

    /* give back our minor */
    usb_deregister_dev(interface, &skel_class);

    /* decrement our usage count */
    kref_put(&dev->kref);

    up(&disconnect_sem);
}
```

Figure 1: Using a lock to ensure safe access to `kref_put`

```
void kref_put(struct kref *kref)
{
    if (atomic_dec_and_test
              (&kref->refcount))
        kref->release(kref);
}
```

This function decrements the value stored in the `refcount` variable, and if the result is zero, this was the last reference to the structure, so the function stored in the `release` variable is called to clean up the memory used by this structure.

4 kref vs. kobject

This paper has focused on on how `struct kref` works, and ignored `struct kobject`. For the most part, both structures work identically, with the following minor differences:

- `struct kobject` does not contain a `release` function. When a `struct kobject`'s last reference count is decremented, the release function of the `struct kset` that is associated with the `struct kobject` is called. For more details on how `struct kobject` and `struct kset` is related, please see [3].
- A `struct kobject` can be initialized with two different functions, `kobject_register` or `kobject_init`. `kobject_register` calls `kobject_init` and then calls `kobject_add` to add the kobject to the sysfs hierarchy. If a `struct kobject` is to not be used within the sysfs hierarchy, then `kobject_add` should never be called.
- A `struct kobject` can have its reference count incremented with a call to `kobject_get` and decremented with a call to `kobject_put`. But if the kobject was initialized with the sysfs core with a call to either `kobject_add` or `kobject_register`, then it needs to be removed from it with a call to `kobject_del`, which will also call `kobject_put` on the `struct kobject`. After a `struct kobject` has had `kobject_del` called for it, the `kboject_get` function can not be called on the variable without having a previous reference count already on the variable. This is the same as the previously mentioned issue for calling `kref_put` without serializing the access.
- Before using a `struct kobject`, the structure must be initialized to zero by using `memset` before `kobject_init` or `kobject_register` is called. If not, a warning will be printed out to the syslog.

5 Future

In future releases of the Linux kernel, the `struct kobject` will probably loose its internal reference count and use the `struct kref` instead. If this happens, `struct kref` might have to be changed in order to support passing the `release` callback as a parameter to the `kref_put` function, in order to save the storage size of the function pointer from the structure.

Other kernel uses of a `atomic_t` variable will probably be converted to use the `struct kref` interface instead of providing their own logic to handle reference counting.

6 Legal Statement

References

[1] Greg Kroah-Hartman. The Driver Model Core, part 1. `http://www.linuxjournal.com/modules.php?op=modload&name=NS-lj-issues/is%sue110&file=6717s2`, June 2003.

[2] Linux Weekly News - Driver porting: Device model overview. `http://lwn.net/Articles/31185/`.

[3] Linux Weekly News - The zen of kobjects. `http://lwn.net/Articles/51437/`.

[4] Patrick Mochel. The Linux Kernel Device Model. In *Linux.conf.au*, Perth, Australia, January 2003.

Proceedings of the Linux Symposium

Volume Two

July 21st–24th, 2004
Ottawa, Ontario
Canada