krefs allow you to add reference counters to your objects. If you have objects that are used in multiple places and passed around, and you don't have refcounts, your code is almost certainly broken. If you want refcounts, krefs are the way to go.

To use a kref, add one to your data structures like:

```
struct my_data
{
	.
	.
	struct kref refcount;
	.
	.
};
```

The kref can occur anywhere within the data structure.

You must initialize the kref after you allocate it. To do this, call kref_init as so:

```
     struct my_data *data;

     data = kmalloc(sizeof(*data), GFP_KERNEL);
     if (!data)
            return -ENOMEM;
     kref_init(&data->refcount);
```

This sets the refcount in the kref to 1.

Once you have an initialized kref, you must follow the following rules:

1) If you make a non-temporary copy of a pointer, especially if it can be passed to another thread of execution, you must increment the refcount with kref_get() before passing it off:
```
	kref_get(&data->refcount);
```
   If you already have a valid pointer to a kref-ed structure (the refcount cannot go to zero) you may do this without a lock.

2) When you are done with a pointer, you must call kref_put():
```
	kref_put(&data->refcount, data_release);
```
   If this is the last reference to the pointer, the release routine will be called. If the code never tries to get a valid pointer to a kref-ed structure without already holding a valid pointer, it is safe to do this without a lock.

3) If the code attempts to gain a reference to a kref-ed structure without already holding a valid pointer, it must serialize access where a kref_put() cannot occur during the kref_get(), and the structure must remain valid during the kref_get().

For example, if you allocate some data and then pass it to another thread to process:

```
void data_release(struct kref *ref)
{
        struct my_data *data = container_of(ref, struct my_data, refcount);
        kfree(data);
}

void more_data_handling(void *cb_data)
{
        struct my_data *data = cb_data;
        .
        . do stuff with data here
        .
        kref_put(&data->refcount, data_release);
}

int my_data_handler(void)
{
        int rv = 0;
        struct my_data *data;
        struct task_struct *task;
        data = kmalloc(sizeof(*data), GFP_KERNEL);
        if (!data)
                return -ENOMEM;
        kref_init(&data->refcount);

        kref_get(&data->refcount);
        task = kthread_run(more_data_handling, data, "more_data_handling");
        if (task == ERR_PTR(-ENOMEM)) {
                rv = -ENOMEM;
                goto out;
        }

        .
        . do stuff with data here
        .
 out:
        kref_put(&data->refcount, data_release);
        return rv;
}
```

This way, it doesn't matter what order the two threads handle the
data, the kref_put() handles knowing when the data is not referenced
any more and releasing it. The kref_get() does not require a lock,
since we already have a valid pointer that we own a refcount for. The
put needs no lock because nothing tries to get the data without
already holding a pointer.

Note that the "before" in rule 1 is very important. You should never
do something like:

```
        task = kthread_run(more_data_handling, data, "more_data_handling");
        if (task == ERR_PTR(-ENOMEM)) {
                rv = -ENOMEM;
                goto out;
        } else
```

```
                /* BAD BAD BAD - get is after the handoff */
                kref_get(&data->refcount);
```

Don't assume you know what you are doing and use the above construct. First of all, you may not know what you are doing. Second, you may know what you are doing (there are some situations where locking is involved where the above may be legal) but someone else who doesn't know what they are doing may change the code or copy the code. It's bad style. Don't do it.

There are some situations where you can optimize the gets and puts. For instance, if you are done with an object and enqueuing it for something else or passing it off to something else, there is no reason to do a get then a put:

```
        /* Silly extra get and put */
        kref_get(&obj->ref);
        enqueue(obj);
        kref_put(&obj->ref, obj_cleanup);
```

Just do the enqueue. A comment about this is always welcome:

```
        enqueue(obj);
        /* We are done with obj, so we pass our refcount off
           to the queue.  DON'T TOUCH obj AFTER HERE! */
```

The last rule (rule 3) is the nastiest one to handle. Say, for instance, you have a list of items that are each kref-ed, and you wish to get the first one. You can't just pull the first item off the list and kref_get() it. That violates rule 3 because you are not already holding a valid pointer. You must add a mutex (or some other lock). For instance:

```
static DEFINE_MUTEX(mutex);
static LIST_HEAD(q);
struct my_data
{
        struct kref      refcount;
        struct list_head link;
};

static struct my_data *get_entry()
{
        struct my_data *entry = NULL;
        mutex_lock(&mutex);
        if (!list_empty(&q)) {
                entry = container_of(q.next, struct my_q_entry, link);
                kref_get(&entry->refcount);
        }
        mutex_unlock(&mutex);
        return entry;
}

static void release_entry(struct kref *ref)
{
        struct my_data *entry = container_of(ref, struct my_data, refcount);
```

```
        list_del(&entry->link);
        kfree(entry);
}

static void put_entry(struct my_data *entry)
{
        mutex_lock(&mutex);
        kref_put(&entry->refcount, release_entry);
        mutex_unlock(&mutex);
}
```

The kref_put() return value is useful if you do not want to hold the lock during the whole release operation. Say you didn't want to call kfree() with the lock held in the example above (since it is kind of pointless to do so). You could use kref_put() as follows:

```
static void release_entry(struct kref *ref)
{
        /* All work is done after the return from kref_put(). */
}

static void put_entry(struct my_data *entry)
{
        mutex_lock(&mutex);
        if (kref_put(&entry->refcount, release_entry)) {
                list_del(&entry->link);
                mutex_unlock(&mutex);
                kfree(entry);
        } else
                mutex_unlock(&mutex);
}
```

This is really more useful if you have to call other routines as part of the free operations that could take a long time or might claim the same lock. Note that doing everything in the release routine is still preferred as it is a little neater.

Corey Minyard <minyard@acm.org>

A lot of this was lifted from Greg Kroah-Hartman's 2004 OLS paper and presentation on krefs, which can be found at:

http://www.kroah.com/linux/talks/ols_2004_kref_paper/Reprint-Kroah-Hartman-OLS2004.pdf
and:
  http://www.kroah.com/linux/talks/ols_2004_kref_talk/