Linux内核部件分析

记录生命周期的kref

[日期：2011-10-06] 来源：Linux社区 作者：qb_2008 [字体：大 中 小]

kref是一个引用计数器，它被嵌套进其它的结构中，记录所嵌套结构的引用计数，并在计数清零时调用相应的清理函数。kref的原理和实现都非常简单，但要想用好却不容易，或者说kref被创建就是为了跟踪复杂情况下地结构引用销毁情况。所以这里先介绍kref的实现，再介绍其使用规则。

kref的头文件在include/linux/kref.h，实现在lib/kref.c。闲话少说，上代码。

```
1. struct kref {
2.     atomic_t refcount;
3. };
```

可以看到，kref的结构中就包含一个atomic_t类型的计数值。atomic_t是原子类型，对其操作都要求是原子执行的，有专门的原子操作API执行，即使在多处理器间也保持原子性。使用atomic_t类型充当计数值，就省去了加锁去锁的过程。

```
1. void kref_set(struct kref *kref, int num)
2. {
3.     atomic_set(&kref->refcount, num);
4.     smp_mb();
5. }
```

kref_set 设置kref的初始计数值。具体计数值设置由原子操作atomic_set完成。之后还有一个smp_mb()是为了增加内存屏障，保证这一写操作会在之后的读写操作完成之前完成。

```
1. void kref_init(struct kref *kref)
2. {
3.     kref_set(kref, 1);
4. }
```

kref_init 初始化kref的计数值为1。

```
1. void kref_get(struct kref *kref)
2. {
3.     WARN_ON(!atomic_read(&kref->refcount));
4.     atomic_inc(&kref->refcount);
5.     smp_mb__after_atomic_inc();
6. }
```

kref_get递增kref的计数值。

```
1. int kref_put(struct kref *kref, void (*release)(struct kref *kref))
2. {
3.     WARN_ON(release == NULL);
4.     WARN_ON(release == (void (*)(struct kref *))kfree);
5.
6.     if (atomic_dec_and_test(&kref->refcount)) {
7.         release(kref);
8.         return 1;
9.     }
10.    return 0;
11. }
```

kref_put递减kref的计数值，如果计数值减为0，说明kref所指向的结构生命周期结束，会执行release释放函数。

所以说kref的API很简单，kref_init和kref_set基本都是初始时才会用到，平时常用的就是kref_get和kref_put。一旦在kref_put时计数值清零，立即调用结束函数。

kref设计得如此简单，是为了能灵活地用在各种结构的生命周期管理中。要用好它可不简单，好在Documentation/kref.txt中为我们总结了一些使用规则，下面简单翻译一下。

对于那些用在多种场合，被到处传递的结构，如果没有引用计数，bug几乎总是肯定的事。所以我们需要kref。kref允许我们在已有的结构中方便地添加引用计数。

你可以以如下方式添加kref到你的数据结构中：

```
1. struct my_data {
2.     ...
3.     struct kref refcount;
4.     ...
5. };
```

kref可以出现在你结构中的任意位置。

在分配kref后你必须初始化它，可以调用kref_init，把kref计数值初始为1。

```
1. struct my_data *data;
2.
3. data = kmalloc(sizeof(*data), GFP_KERNEL);
4. if(!data)
5.     return -ENOMEM;
6. kref_init(&data->refcount);
```

初始化之后，kref的使用应该遵循以下三条规则：

1） 如果你制造了一个结构指针的非暂时性副本，特别是当这个副本指针会被传递到其它执行线程时，你必须在传递副本指针之前执行kref_get：

```
1. kref_put(&data->refcount);
```

2）当你使用完，不再需要结构的指针，必须执行kref_put。如果这是结构指针的最后一个引用，release函数会被调用。如果代码绝不会在没有拥有引用计数的请求下去调用kref_get，在kref_put时就不需要加锁。

```
1. kref_put(&data->refcount, data_release);
```

3）如果代码试图在还没拥有引用计数的情况下就调用kref_get，就必须串行化kref_put和kref_get的执行。因为很可能在kref_get执行之前或者执行中，kref_put就被调用并把整个结构释放掉了。

例如，你分配了一些数据并把它传递到其它线程去处理：

```
1. void data_release(struct kref *kref)
2. {
3.     struct my_data *data = container_of(kref, struct my_data, refcount);
4.     kree(data);
5. }
6.
7. void more_data_handling(void *cb_data)
8. {
9.     struct my_data *data = cb_data;
10.    .
11.    .  do stuff with data here
12.    .
13.    kref_put(&data->refcount, data_release);
14. }
15.
16. int my_data_handler(void)
17. {
18.    int rv = 0;
19.    struct my_data *data;
20.    struct task_struct *task;
21.    data = kmalloc(sizeof(*data), GFP_KERNEL);
22.     if (!data)
23.        return -ENOMEM;
24.    kref_init(&data->refcount);
25.    kref_get(&data->refcount);
26.    task = kthread_run(more_data_handling, data, "more_data_handling");
27.    if (task == ERR_PTR(-ENOMEM)){
28.         rv = -ENOMEM;
29.         goto out;
30.    }
31.    .
32.    .  do stuff with data here
33.    .
34. out:
35.    kref_put(&data->refcount, data_release);
36.    return rv;
37. }
```

这样做，无论两个线程的执行顺序是怎样的都无所谓，kref_put知道何时数据不再有引用计数，可以被销毁。kref_get()调用不需要加锁，因为在my_data_handler中调用kref_get时已经拥有一个引用。同样地原因，kref_put也不需要加锁。

要注意规则一中的要求，必须在传递指针之前调用kref_get。决不能写下面的代码：

```
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);
```

不要认为自己在使用上面的代码时知道自己在做什么。首先，你可能并不知道你在做什么。其次，你可能知道你在做什么（在部分加锁情况下上面的代码也是正确的），但一些修改或者复制你代码的人并不知道你在做什么。这是一种坏的使用方式。

当然在部分情况下也可以优化对get和put的使用。例如，你已经完成了对这个数据的处理，并要把它传递给其它线程，就不需要再做多余的get和put了。

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

只需要做enqueue操作即可，可以在其后加一条注释。

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

第三条规则是处理起来最麻烦的。例如，你有一列数据，每条数据都有kref计数，你希望获取第一条数据。但你不能简单地把第一条数据从链表中取出并调用kref_get。这违背了第三条，在调用kref_get前你并没有一个引用。你需要增加一个mutex（或者其它锁）。

```
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);
}
```

如果你不想在整个释放过程中都加锁，kref_put的返回值就有用了。例如你不想在加锁情况下调用kfree，你可以如下使用kref_put。

```
static void release_entry(struct kref *ref)
{

}

static void put_entry(struct my_data *entry)
{
```

```
8.  mutex_lock(&mutex);
9.  if(kref_put(&entry->refcount, release_entry)){
10.     list_del(&entry->link);
11.     mutex_unlock(&mutex);
12.     kfree(entry);
13. }
14. else
15.     mutex_unlock(&mutex);
16. }
```

如果你在撤销结构的过程中需要调用其它的需要较长时间的函数，或者函数也可能要获取同样地互斥锁，这样做就很有用了。但要注意在release函数中做完撤销工作会使代码看起来更整洁。

【内容导航】

Linux内核的学习方法 Linux根目录下主要目录功能说明及常用分区方案

相关资讯 Linux内核

Linux内核Git源码树中的代码已达 (今 20:48)
Linux 5.4.7 / 4.19.92 / 4.14.161 (01月01日)
Linux内核将用Rust编程语言编写？ (09/03/2019 12:06:17)
Linux内核将很快默认情况启用"- (05/11/2019 13:43:07)
Linux内核正在努力实现快速高效的I (02/15/2019 14:51:33)
Linux内核的冷热缓存 (01/27/2019 19:10:52)

本文评论 查看全部评论 (5)

表情： 姓名： 匿名 匿名 字数 0

同意评论声明 请登录

评论声明

- 尊重网上道德，遵守中华人民共和国的各项有关法律法规
- 承担一切因您的行为而直接或间接导致的民事或刑事法律责任
- 本站管理人员有权保留或删除其管辖留言中的任意内容
- 本站有权在网站内转载或引用您的评论
- 参与本评论即表明您已经阅读并接受上述条款

AlexXue 发表于 2018/6/22 11:29:47 第 5 楼

好吧，没有问题，当我没说

回复 支持 (0) 反对 (0)

AlexXue 发表于 2018/6/22 9:05:51 第 4 楼

作者，我认为你的__list_add函数有问题。麻烦画图分析一下。

回复 支持 (0) 反对 (0)

AlexXue 发表于 2018/6/20 10:07:20 第 3 楼

作者你好关于__list_add这个函数，我画图分析之后发现存在问题，烦请您贴图分析一下，感谢。

回复 支持 (0) 反对 (0)

lzxname 发表于 2014/11/4 9:48:24 第 2 楼

好东西。留一笔。

回复 支持 (14) 反对 (11)

lzxname 发表于 2014/11/4 9:43:13 第 1 楼

好东西啊啊。。。

回复 支持 (8) 反对 (13)