Linux内核部件分析

更强的链表klist

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

前面我们说到过list_head，这是linux中通用的链表形式，双向循环链表，功能强大，实现简单优雅。可如果您认为list_head就是链表的极致，应该在linux链表界一统天下，那可就错了。据我所知，linux内核代码中至少还有两种链表能占有一席之地。一种就是hlist，一种就是本节要介绍的klist。虽然三者不同，但hlist和klist都可以看成是从list_head中发展出来的，用于特殊的链表使用情景。hlist是用于哈希表中。众所周知，哈希表主要就是一个哈希数组，为了解决映射冲突的问题，常常把哈希数组的每一项做成一个链表，这样有多少重复的都可以链进去。但哈希数组的项很多，list_head的话每个链表头都需要两个指针的空间，在稀疏的哈希表中实在是一种浪费，于是就发明了hlist。hlist有两大特点，一是它的链表头只需要一个指针，二是它的每一项都可以找到自己的前一节点，也就是说它不再循环，但仍是双向。令人不解的是，hlist的实现太绕了，比如它明明可以直接指向前一节点，却偏偏指向指针地址，还是前一节点中指向后一节点的指针地址。即使这种设计在实现时占便宜，但它理解上带来的不便已经远远超过实现上带来的小小便利。

同hlist一样，klist也是为了适应某类特殊情形的要求。考虑一个被简化的情形，假设一些设备被链接在设备链表中，一个线程命令卸载某设备，即将其从设备链表中删除，但这时该设备正在使用中，这时就出现了冲突。当前可以设置临界区并加锁，但因为使用一个设备而锁住整个设备链表显然是不对的；又或者可以从设备本身做文章，让线程阻塞，这当然也可以。但我们上节了解了kref，就该知道linux对待这种情况的风格，给它一个引用计数kref，等计数为零就删除。klist就是这么干的，它把kref直接保存在了链表节点上。之前说到有线程要求删除设备，之前的使用仍存在，所以不能实际删除，但不应该有新的应用访问到该设备。klist就提供了一种让节点在链表上隐身的方法。下面还是来看实际代码吧。

klist的头文件是include/linux/klist.h，实现在lib/klist.c。

```
1. struct klist_node;
2. struct klist {
3.     spinlock_t      k_lock;
4.     struct list_head    k_list;
5.     void            (*get)(struct klist_node *);
6.     void            (*put)(struct klist_node *);
7. } __attribute__ ((aligned (4)));
8.
9. #define KLIST_INIT(_name, _get, _put)                   \
10.    { .k_lock   = __SPIN_LOCK_UNLOCKED(_name.k_lock),       \
11.      .k_list   = LIST_HEAD_INIT(_name.k_list),         \
12.      .get      = _get,                     \
13.      .put      = _put, }
14.
15. #define DEFINE_KLIST(_name, _get, _put)                 \
16.     struct klist _name = KLIST_INIT(_name, _get, _put)
17.
18. extern void klist_init(struct klist *k, void (*get)(struct klist_node *),
19.              void (*put)(struct klist_node *));
20.
21. struct klist_node {
22.     void            *n_klist;   /* never access directly */
23.     struct list_head    n_node;
24.     struct kref     n_ref;
25. };
```

可以看到，klist的链表头是struct klist结构，链表节点是struct klist_node结构。先看struct klist，除了包含链表需要的k_list，还有用于加锁的k_lock。剩余的get()和put()函数是用于struct klist_node嵌入在更大的结构中，这样在节点初始时调用get()，在节点删除时调用put()，以表示链表中存在对结构的引用。再看struct klist_node，除了链表需要的n_node，还有一个引用计数n_ref。还有一个比较特殊的指针n_klist，n_klist是指向链表头struct klist的，但它的第0位用来表示是否该节点已被请求删除，如果已被请求删除则在链表循环时是看不到这一节点的，循环函数将其略过。现在你明白为什么非要在struct klist的定义后加上__attribute__((aligned(4)))。不过说实话这样在x86下仍然不太保险，但linux选择了相信gcc，毕竟是多年的战友和兄弟了，相互知根知底。

看过这两个结构，想必大家已经较为清楚了，下面就来看看它们的实现。

```
1. /*
2.  * Use the lowest bit of n_klist to mark deleted nodes and exclude
3.  * dead ones from iteration.
4.  */
5. #define KNODE_DEAD      1LU
6. #define KNODE_KLIST_MASK    ~KNODE_DEAD
7.
```

```
8. static struct klist *knode_klist(struct klist_node *knode)
9. {
10.     return (struct klist *)
11.         ((unsigned long)knode->n_klist & KNODE_KLIST_MASK);
12. }
13.
14. static bool knode_dead(struct klist_node *knode)
15. {
16.     return (unsigned long)knode->n_klist & KNODE_DEAD;
17. }
18.
19. static void knode_set_klist(struct klist_node *knode, struct klist *klist)
20. {
21.     knode->n_klist = klist;
22.     /* no knode deserves to start its life dead */
23.     WARN_ON(knode_dead(knode));
24. }
25.
26. static void knode_kill(struct klist_node *knode)
27. {
28.     /* and no knode should die twice ever either, see we're very humane */
29.     WARN_ON(knode_dead(knode));
30.     *(unsigned long *)&knode->n_klist |= KNODE_DEAD;
31. }
```

前面的四个函数都是内部静态函数，帮助API实现的。knode_klist()是从节点找到链表头。knode_dead()是检查该节点是否已被请求删除。

knode_set_klist设置节点的链表头。knode_kill将该节点请求删除。细心的话大家会发现这四个函数是对称的，而且都是操作节点的内部函数。

```
1. void klist_init(struct klist *k, void (*get)(struct klist_node *),
2.         void (*put)(struct klist_node *))
3. {
4.     INIT_LIST_HEAD(&k->k_list);
5.     spin_lock_init(&k->k_lock);
6.     k->get = get;
7.     k->put = put;
8. }
```

klist_init，初始化klist。

```
1. static void add_head(struct klist *k, struct klist_node *n)
2. {
3.     spin_lock(&k->k_lock);
4.     list_add(&n->n_node, &k->k_list);
5.     spin_unlock(&k->k_lock);
6. }
7.
8. static void add_tail(struct klist *k, struct klist_node *n)
9. {
10.     spin_lock(&k->k_lock);
11.     list_add_tail(&n->n_node, &k->k_list);
12.     spin_unlock(&k->k_lock);
13. }
14.
15. static void klist_node_init(struct klist *k, struct klist_node *n)
16. {
17.     INIT_LIST_HEAD(&n->n_node);
18.     kref_init(&n->n_ref);
19.     knode_set_klist(n, k);
20.     if (k->get)
21.         k->get(n);
22. }
```

又是三个内部函数，add_head()将节点加入链表头，add_tail()将节点加入链表尾，klist_node_init()是初始化节点。注意在节点的引用计数初始化时，因为引用计数变为1，所以也要调用相应的get()函数。

```
1. void klist_add_head(struct klist_node *n, struct klist *k)
2. {
3.     klist_node_init(k, n);
4.     add_head(k, n);
5. }
6.
```

```
7. void klist_add_tail(struct klist_node *n, struct klist *k)
8. {
9.     klist_node_init(k, n);
10.    add_tail(k, n);
11. }
```

klist_add_head()将节点初始化，并加入链表头。

klist_add_tail()将节点初始化，并加入链表尾。

它们正是用上面的三个内部函数实现的，可见linux内核中对函数复用有很强的执念，其实这里add_tail和add_head是不用的，纵观整个文件，也只有klist_add_head()和klist_add_tail()对它们进行了调用。

```
1. void klist_add_after(struct klist_node *n, struct klist_node *pos)
2. {
3.     struct klist *k = knode_klist(pos);
4.
5.     klist_node_init(k, n);
6.     spin_lock(&k->k_lock);
7.     list_add(&n->n_node, &pos->n_node);
8.     spin_unlock(&k->k_lock);
9. }
10.
11. void klist_add_before(struct klist_node *n, struct klist_node *pos)
12. {
13.     struct klist *k = knode_klist(pos);
14.
15.     klist_node_init(k, n);
16.     spin_lock(&k->k_lock);
17.     list_add_tail(&n->n_node, &pos->n_node);
18.     spin_unlock(&k->k_lock);
19. }
```

klist_add_after()将节点加到指定节点后面。

klist_add_before()将节点加到指定节点前面。

这两个函数都是对外提供的API。在list_head中都没有看到有这种API，所以说需求决定了接口。虽说只有一步之遥，klist也不愿让外界介入它的内部实现。

之前出现的API都太常见了，既没有使用引用计数，又没有跳过请求删除的节点。所以klist的亮点在下面，klist链表的遍历。

```
1. struct klist_iter {
2.     struct klist     *i_klist;
3.     struct klist_node    *i_cur;
4. };
5.
6.
7. extern void klist_iter_init(struct klist *k, struct klist_iter *i);
8. extern void klist_iter_init_node(struct klist *k, struct klist_iter *i,
9.                  struct klist_node *n);
10. extern void klist_iter_exit(struct klist_iter *i);
11. extern struct klist_node *klist_next(struct klist_iter *i);
```

以上就是链表遍历需要的辅助结构struct klist_iter，和遍历用到的四个函数。

```
1. struct klist_waiter {
2.     struct list_head list;
3.     struct klist_node *node;
4.     struct task_struct *process;
5.     int woken;
6. };
7.
8. static DEFINE_SPINLOCK(klist_remove_lock);
9. static LIST_HEAD(klist_remove_waiters);
10.
11. static void klist_release(struct kref *kref)
12. {
13.     struct klist_waiter *waiter, *tmp;
14.     struct klist_node *n = container_of(kref, struct klist_node, n_ref);
15.
16.     WARN_ON(!knode_dead(n));
17.     list_del(&n->n_node);
18.     spin_lock(&klist_remove_lock);
19.     list_for_each_entry_safe(waiter, tmp, &klist_remove_waiters, list) {
```

```
20.         if (waiter->node != n)
21.             continue;
22.
23.         waiter->woken = 1;
24.         mb();
25.         wake_up_process(waiter->process);
26.         list_del(&waiter->list);
27.     }
28.     spin_unlock(&klist_remove_lock);
29.     knode_set_klist(n, NULL);
30. }
31.
32. static int klist_dec_and_del(struct klist_node *n)
33. {
34.     return kref_put(&n->n_ref, klist_release);
35. }
36.
37. static void klist_put(struct klist_node *n, bool kill)
38. {
39.     struct klist *k = knode_klist(n);
40.     void (*put)(struct klist_node *) = k->put;
41.
42.     spin_lock(&k->k_lock);
43.     if (kill)
44.         knode_kill(n);
45.     if (!klist_dec_and_del(n))
46.         put = NULL;
47.     spin_unlock(&k->k_lock);
48.     if (put)
49.         put(n);
50. }
51.
52. /**
53.  * klist_del - Decrement the reference count of node and try to remove.
54.  * @n: node we're deleting.
55.  */
56. void klist_del(struct klist_node *n)
57. {
58.     klist_put(n, true);
59. }
```

以上的内容乍一看很难理解，其实都是klist实现必须的。因为使用kref动态删除，自然需要一个计数降为零时调用的函数klist_release。

klist_dec_and_del()就是对kref_put()的包装，起到减少节点引用计数的功能。

至于为什么会出现一个新的结构struct klist_waiter，也很简单。之前说有线程申请删除某节点，但节点的引用计数仍在，所以只能把请求删除的线程阻塞，就是用struct klist_waiter阻塞在klist_remove_waiters上。所以在klist_release()调用时还要将阻塞的线程唤醒。knode_kill()将节点设为已请求删除。而且还会调用put()函数。

释放引用计数是调用klist_del()，它通过内部函数klist_put()完成所需操作：用knode_kill()设置节点为已请求删除，用klist_dec_and_del()释放引用，调用可能的put()函数。

```
1. /**
2.  * klist_remove - Decrement the refcount of node and wait for it to go away.
3.  * @n: node we're removing.
4.  */
5. void klist_remove(struct klist_node *n)
6. {
7.     struct klist_waiter waiter;
8.
9.     waiter.node = n;
10.    waiter.process = current;
11.    waiter.woken = 0;
12.    spin_lock(&klist_remove_lock);
13.    list_add(&waiter.list, &klist_remove_waiters);
14.    spin_unlock(&klist_remove_lock);
15.
16.    klist_del(n);
17.
18.    for (;;) {
```

```
19.         set_current_state(TASK_UNINTERRUPTIBLE);
20.         if (waiter.woken)
21.             break;
22.         schedule();
23.     }
24.     __set_current_state(TASK_RUNNING);
25. }
```

klist_remove()不但会调用klist_del()减少引用计数，还会一直阻塞到节点被删除。这个函数才是请求删除节点的线程应该调用的。

```
1. int klist_node_attached(struct klist_node *n)
2. {
3.     return (n->n_klist != NULL);
4. }
```

klist_node_attached()检查节点是否被包含在某链表中。

以上是klist的链表初始化，节点加入，节点删除函数。下面是klist链表遍历函数。

```
1. struct klist_iter {
2.     struct klist        *i_klist;
3.     struct klist_node   *i_cur;
4. };
5.
6.
7. extern void klist_iter_init(struct klist *k, struct klist_iter *i);
8. extern void klist_iter_init_node(struct klist *k, struct klist_iter *i,
9.                  struct klist_node *n);
10. extern void klist_iter_exit(struct klist_iter *i);
11. extern struct klist_node *klist_next(struct klist_iter *i);
```

klist的遍历有些复杂，因为它考虑到了在遍历过程中节点删除的情况，而且还要忽略那些已被删除的节点。宏实现已经无法满足要求，迫不得已，只能用函数实现，并用struct klist_iter记录中间状态。

```
1. void klist_iter_init_node(struct klist *k, struct klist_iter *i,
2.              struct klist_node *n)
3. {
4.     i->i_klist = k;
5.     i->i_cur = n;
6.     if (n)
7.         kref_get(&n->n_ref);
8. }
9.
10. void klist_iter_init(struct klist *k, struct klist_iter *i)
11. {
12.     klist_iter_init_node(k, i, NULL);
13. }
```

klist_iter_init_node()是从klist中的某个节点开始遍历，而klist_iter_init()是从链表头开始遍历的。

但你又要注意，klist_iter_init()和klist_iter_init_node()的用法又不同。klist_iter_init_node()可以在其后直接对当前节点进行访问，也可以调用klist_next()访问下一节点。而klist_iter_init()只能调用klist_next()访问下一节点。或许klist_iter_init_node()的本意不是从当前节点开始，而是从当前节点的下一节点开始。

```
1. static struct klist_node *to_klist_node(struct list_head *n)
2. {
3.     return container_of(n, struct klist_node, n_node);
4. }
```

```
1. struct klist_node *klist_next(struct klist_iter *i)
2. {
3.     void (*put)(struct klist_node *) = i->i_klist->put;
4.     struct klist_node *last = i->i_cur;
5.     struct klist_node *next;
6.
7.     spin_lock(&i->i_klist->k_lock);
8.
9.     if (last) {
10.         next = to_klist_node(last->n_node.next);
11.         if (!klist_dec_and_del(last))
12.             put = NULL;
13.     } else
14.         next = to_klist_node(i->i_klist->k_list.next);
```

```
15.
16.     i->i_cur = NULL;
17.     while (next != to_klist_node(&i->i_klist->k_list)) {
18.         if (likely(!knode_dead(next))) {
19.             kref_get(&next->n_ref);
20.             i->i_cur = next;
21.             break;
22.         }
23.         next = to_klist_node(next->n_node.next);
24.     }
25.
26.     spin_unlock(&i->i_klist->k_lock);
27.
28.     if (put && last)
29.         put(last);
30.     return i->i_cur;
31. }
```

klist_next()是将循环进行到下一节点。实现中需要注意两点问题：1、加锁，根据经验，单纯对某个节点操作不需要加锁，但对影响整个链表的操作需要加自旋锁。比如之前klist_iter_init_node()中对节点增加引用计数，就不需要加锁，因为只有已经拥有节点引用计数的线程才会特别地从那个节点开始。而之后klist_next()中则需要加锁，因为当前线程很可能没有引用计数，所以需要加锁，让情况固定下来。这既是保护链表，也是保护节点有效。符合kref引用计数的使用原则。2、要注意，虽然在节点切换的过程中是加锁的，但切换完访问当前节点时是解锁的，中间可能有节点被删除（这个通过spin_lock就可以搞定），也可能有节点被请求删除，这就需要注意。首先要忽略链表中已被请求删除的节点，然后在减少前一个节点引用计数时，可能就把前一个节点删除了。这里之所以不调用klist_put()，是因为本身已处于加锁状态，但仍要有它的实现。这里的实现和klist_put()中类似，代码不介意在加锁状态下唤醒另一个线程，但却不希望在加锁状态下调用put()函数，那可能会涉及释放另一个更大的结构。

```
1. void klist_iter_exit(struct klist_iter *i)
2. {
3.     if (i->i_cur) {
4.         klist_put(i->i_cur, false);
5.         i->i_cur = NULL;
6.     }
7. }
```

klist_iter_exit()，遍历结束函数。在遍历完成时调不调无所谓，但如果想中途结束，就一定要调用klist_iter_exit()。

klist主要用于设备驱动模型中，为了适应那些动态变化的设备和驱动，而专门设计的链表。klist并不通用，但它真的很新奇。 我看到它时，震惊于链表竟然可以专门异化成这种样子。如果你是松耦合的结构，如果你手下净是些桀骜不驯的家伙，那么不要只考虑kref，你可能还需要klist。

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