

Author:joyce

Email:jfferyjoyce.org@gmail.com

Date:2012 08 29

一 概述

大多数内核子系统都是相互独立的,因此某个子系统可能对其它子系统产生的事件感兴趣。为了满足这个需求,也即是让某个子系统在发生某个事件时通知其它的子系统,Linux 内核提供了通知链的机制。通知链表只能够在内核的子系统之间使用,而不能在内核与用户空间之间进行事件的通知。通知链表是一个函数链表,链表上的每一个节点都注册了一个函数。当某个事情发生时,链表上所有节点对应的函数就会被执行。所以对于通知链表来说有一个通知方与一个接收方。在通知这个事件时所运行的函数由被通知方决定,实际上也即是被通知方注册了某个函数,在发生某个事件时这些函数就得到执行。

源码路径: include/linux/notifier.h

二 notifier 数据结构类型

目前 Linux 内核支持四种类型的通知链表机制,它们的定义如下,为了更好的理解我将每种类型的通知链理解成一个广播接收者,产生事件的那一端理解成广播发送者

```
/*
 *atomic notifier chains: Chain callbacks run in interrupt/atomic
 *context. Callouts are not allowed to block.
 *中断,原子上下文,不允许阻塞操作
 */
struct atomic_notifier_head {
    spinlock_t lock;
    struct notifier_block __rcu *head;
};

/*
 *Blocking notifier chains: Chain callbacks run in process context.
 *Callouts are allowed to block.
 *进程上下文,允许阻塞
 */
struct blocking_notifier_head {
    struct rw_semaphore rwsem; /*读取者与写入者信号量*/
    struct notifier_block __rcu *head;
};

/*
 *Raw notifier chains: There are no restrictions on callbacks,
 *registration, or unregistration. All locking and protection
 *must be provided by the caller.
 *原始通知链: 对通知链元素的回调函数没有任何限制,所有锁和保护机制都由调用者维护
 */
struct raw_notifier_head {
    struct notifier_block __rcu *head;
};

/*
 *SRCU notifier chains: A variant of blocking notifier chains, with
 *the same restrictions.
 *Blocking notifier chains 的一种变体,运行在进程上下文允许阻塞
 */
```

```
struct srcu_notifier_head {
    struct mutex mutex;
    struct srcu_struct srcu;
    struct notifier_block __rcu *head;
};
```

三 notifier 系统 API

上面每一个通知链中都包含了一个核心的数据结构,该结构由广播发送者定义并调用系统 API 向广播接收者注册

```
struct notifier_block {
    int (*notifier_call)(struct notifier_block *, unsigned long, void *);
    struct notifier_block __rcu *next; /*指向广播接收者中对应的数据结构*/
    int priority; /*指名本通知链在链表中的优先级*/
};
```

其中 notifier_call 是通知链要执行的函数指针,当广播发送者发出相应的通知后,并且由相对应的广播接受者接收到该通知,notifier_call 将会被调用

系统 API

Linux 内核为我们提供了 4 中 API 分别用于定义并初始化每一种类型的通知链,具体如下

动态定义并初始化原子型通知链 (原子型广播接收者)

```
#define ATOMIC_INIT_NOTIFIER_HEAD(name) do { \
    spin_lock_init(&(name)->lock); \
    (name)->head = NULL; \
} while (0)
```

动态定义并初始化阻塞型通知链 (阻塞型广播接收者)

```
#define BLOCKING_INIT_NOTIFIER_HEAD(name) do { \
    init_rwsem(&(name)->rwsem); \
    (name)->head = NULL; \
} while (0)
```

动态定义并初始化原始型通知链 (原始型广播接收者)

```
#define RAW_INIT_NOTIFIER_HEAD(name) do { \
    (name)->head = NULL; \
} while (0)
```

动态定义并初始化 srcu 型通知链 (srcu 型广播接收者) 注意其中 srcu 型的通知链只能被动态申请初始化

```
/* srcu_notifier_heads must be initialized and cleaned up dynamically */
extern void srcu_init_notifier_head(struct srcu_notifier_head *nh);
#define srcu_cleanup_notifier_head(name) \
    cleanup_srcu_struct(&(name)->srcu);
```

静态定义并初始化原子型通知链

```
#define ATOMIC_NOTIFIER_INIT(name) { \
    .lock = __SPIN_LOCK_UNLOCKED(name.lock), \
    .head = NULL } \
```

静态定义并初始化阻塞型通知链

```
#define BLOCKING_NOTIFIER_INIT(name) { \
    .rwsem = __RWSEM_INITIALIZER((name).rwsem), \
    .head = NULL } \
```

静态定义并初始化原始型通知链

```
#define RAW_NOTIFIER_INIT(name) { \
    .head = NULL }
```

对应上述的每一种类型的通知链(广播接收者),Linux 内核提供了相对应的 API 向每一种类型的通知链(广播接收者)注册和注销一个通知或(广播),具体的实现在 kernel/notifier.c 中

向广播接收者注册广播或者说向通知链注册通知

```
int atomic_notifier_chain_register(struct atomic_notifier_head *nh,
    struct notifier_block *n)
```

向广播接收者注销广播或者说向通知链注销通知

```
int atomic_notifier_chain_unregister(struct atomic_notifier_head *nh,
    struct notifier_block *n)
```

```
int blocking_notifier_chain_register(struct blocking_notifier_head *nh,
    struct notifier_block *n)
```

```
int blocking_notifier_chain_cond_register(struct blocking_notifier_head
    *nh,
    struct notifier_block *n)
```

```
int blocking_notifier_chain_unregister(struct blocking_notifier_head *nh,
    struct notifier_block *n)
```

```
int raw_notifier_chain_register(struct raw_notifier_head *nh,
    struct notifier_block *n)
```

```
int raw_notifier_chain_unregister(struct raw_notifier_head *nh,
    struct notifier_block *n)
```

```
int srcu_notifier_chain_register(struct srcu_notifier_head *nh,
    struct notifier_block *n)
```

```
int srcu_notifier_chain_unregister(struct srcu_notifier_head *nh,
    struct notifier_block *n)
```

四 notifier 系统 API 源码情景导读

现在对广播的注册(通知向通知链的注册)函数进行分析

```
/**
 *blocking_notifier_chain_register - Add notifier to a blocking notifier
chain
 *@nh: Pointer to head of the blocking notifier chain
 *@n: New entry in notifier chain
 *Adds a notifier to a blocking notifier chain.
 *Must be called in process context.
 *Currently always returns zero.
 */
int blocking_notifier_chain_register(struct blocking_notifier_head *nh,
    struct notifier_block *n)
{
    int ret;

    /*
     * This code gets used during boot-up, when task switching is
     * not yet working and interrupts must remain disabled. At
```

```

    * such times we must not call down_write().
    */
    if (unlikely(system_state == SYSTEM_BOOTING))
        return notifier_chain_register(&nh->head, n);

    down_write(&nh->rwsem); //获取写入者信号量
    ret = notifier_chain_register(&nh->head, n);
    up_write(&nh->rwsem); //释放写入者信号量
    return ret;
}

/*
 *Notifier chain core routines. The exported routines below
 *are layered on top of these, with appropriate locking added.
 */
static int notifier_chain_register(struct notifier_block **nl,
                                   struct notifier_block *n)
{
    while ((*nl) != NULL) { //如果该通知链不为空,则执行下面循环
        if (n->priority > (*nl)->priority)
            break;
        nl = &((*nl)->next);
    }
    n->next = *nl;
    rcu_assign_pointer(*nl, n);
    return 0;
}


```

当有事件发生时,广播接收者调用 `notifier_call_chain` 函数通知(广播)事件的到达,这个函数会遍历 `nl` 指向的通知链中所有的元素,然后依次调用每一个的回调函数,完成通知动作。

```

/**
 * __blocking_notifier_call_chain - Call functions in a blocking notifier
chain
 *@nh: Pointer to head of the blocking notifier chain
 *@val: Value passed unmodified to notifier function
 *@v: Pointer passed unmodified to notifier function
 *@nr_to_call: See comment for notifier_call_chain.
 *@nr_calls: See comment for notifier_call_chain.
 *Calls each function in a notifier chain in turn. The functions
 *run in a process context, so they are allowed to block.
 *If the return value of the notifier can be and'ed
 *with %NOTIFY_STOP_MASK then blocking_notifier_call_chain()
 *will return immediately, with the return value of
 *the notifier function which halted execution.
 *Otherwise the return value is the return value
 *of the last notifier function called.
 */
int __blocking_notifier_call_chain(struct blocking_notifier_head *nh,
                                   unsigned long val, void *v,
                                   int nr_to_call, int *nr_calls)
{
    int ret = NOTIFY_DONE;

```



```

/*
 * We check the head outside the lock, but if this access is
 * racy then it does not matter what the result of the test
 * is, we re-check the list after having taken the lock anyway:
 */
if (rcu_dereference_raw(nh->head)) {
    down_read(&nh->rwsem); //获取读取者信号量
    ret = notifier_call_chain(&nh->head, val, v, nr_to_call,
                             nr_calls);
    up_read(&nh->rwsem); //释放读取者信号量
}
return ret;
}

```

广播接收者调用下面这个函数来告知通知或(广播事件)的到达

```

int blocking_notifier_call_chain(struct blocking_notifier_head *nh,
    unsigned long val, void *v)
{
    return __blocking_notifier_call_chain(nh, val, v, -1, NULL);
}

```

```

/**
 * notifier_call_chain - Informs the registered notifiers about an event.
 * @nl: Pointer to head of the blocking notifier chain
 * @val: 事件类型
 * @v: 用来指向通知链上的函数执行时需要用到的参数,例如当通知一个网卡被注册时,v就指向
    net_device 结构
 * @nr_to_call: Number of notifier functions to be called. Don't care
 *              value of this parameter is -1.
 * @nr_calls: Records the number of notifications sent. Don't care
 *            value of this field is NULL.用于记录发送通知的条数
 * @returns: notifier_call_chain returns the value returned by the
 *           last notifier function called.
 */
static int kprobes_notifier_call_chain(struct notifier_block **nl,
    unsigned long val, void *v,
    int nr_to_call, int *nr_calls)
{
    int ret = NOTIFY_DONE;
    struct notifier_block *nb, *next_nb;

    nb = rcu_dereference_raw(*nl);

    while (nb && nr_to_call) {
        /*读者调用它来获得一个被RCU保护的指针*/
        next_nb = rcu_dereference_raw(nb->next);

        /*这里是不是调用广播发送者的回调函数了?*/
        ret = nb->notifier_call(nb, val, v);
        if (nr_calls)
            (*nr_calls)++;
    }

    if ((ret & NOTIFY_STOP_MASK) == NOTIFY_STOP_MASK)

```

```

        break;
        nb = next_nb;
        nr_to_call--;
    }
    return ret;
}

```

每个被执行的 `notifier_block` 回调函数的返回值定义在 `include/linux/notifier.h` 中

```

#define NOTIFY_DONE      0x0000      /* Don't care */
#define NOTIFY_OK        0x0001      /* Suits me */
#define NOTIFY_STOP_MASK 0x8000      /* Don't call further */
#define NOTIFY_BAD       (NOTIFY_STOP_MASK|0x0002) /* 执行出错 */

```

五 Linux 内核通知链的运行机制

- 1) 广播接收者调用系统 API 定义通知链
- 2) 广播发送者定义 `struct srcu_notifier_head` 结构, 并初始化结构中的通知回调函数和成员变量
- 3) 广播发送者调用系统 API 向广播接收者注册
- 4) 广播发送者, 某事件被出发, 向广播接收这发送一广播
- 5) 广播接收者接收该广播调用系统某 API 对该广播通知所在的链表进行遍历
- 6) 最后广播发送者中的回调函数被执行, 产生具体的动作

六 Linux 内核通知链典型应用

附件代码

