内核通知链

1、背景

在linux内核中，各个子系统之间有很强的相互关系，某些子系统可能对其他子系统产生的事件比较感兴趣。因此内核引入了notifier机制，当然了notifier机制只能用在内核子系统之间，不能用在内核与应用层之间。比如当系统suspend的时候，就会使用到notifier机制来通知系统的内核线程进行suspend。内核实现的notifier机制代码位于kernel/kernel/notifier.c，同时此机制的代码量也不是很多只有600行左右。

2、数据结构

内核使用 `struct notifier_block` 结构代表一个 `notifier`

```
typedef int (*notifier_fn_t)(struct notifier_block *nb,
            unsigned long action, void *data);

struct notifier_block {
    //代表事件发生之后调用的回调函数
    notifier_fn_t notifier_call;
    //用来链接同一类型的notifier
    struct notifier_block __rcu *next;
    // 通知链的优先级, 数字越大优先级越高, 优先执行
    int priority;
};
```

3、通知链的类型

- 原子通知链

仅仅对 `notifier_block` 的封装。 `atomic_notifer_head` 中包含 `spin_lock` 表示不能睡眠通知链元素的回调函数（事件发生要执行的函数）在中断或原子操作上下文中运行，不允许阻塞

```
struct atomic_notifier_head {
    spinlock_t lock;
    struct notifier_block __rcu *head;
};
```

- 可阻塞通知链

包含读写信号量成员`rwsem`，信号量的特点就是运行在进程上下文，还可以睡眠。

```
struct blocking_notifier_head {
    struct rw_semaphore rwsem;
    struct notifier_block __rcu *head;
};
```

- 原始通知链

没有任何限制，需要调用者维护

```
struct raw_notifier_head {
    struct notifier_block __rcu *head;
};
```

- SRCU通知链

是**block notifier chain**的变体，采用**SRCU** （**Sleepable Read-Copy Update**） 代替**rw-semphore**保护**chains**

```
struct srcu_struct {
    short srcu_lock_nesting[2]; /* srcu_read_lock() nesting depth. */
    short srcu_idx;         /* Current reader array element. */
    u8 srcu_gp_running;     /* GP workqueue running? */
    u8 srcu_gp_waiting;     /* GP waiting for readers? */
    struct swait_queue_head srcu_wq;
                    /* Last srcu_read_unlock() wakes GP. */
    struct rcu_head *srcu_cb_head;  /* Pending callbacks: Head. */
    struct rcu_head **srcu_cb_tail; /* Pending callbacks: Tail. */
    struct work_struct srcu_work;   /* For driving grace periods. */
#ifdef CONFIG_DEBUG_LOCK_ALLOC
    struct lockdep_map dep_map;
#endif /* #ifdef CONFIG_DEBUG_LOCK_ALLOC */
};

struct srcu_notifier_head {
    // 通知链可能被多个线程同时访问或修改，所以需要使用互斥锁保证线程安全性
    struct mutex mutex;
    // 提供机制，在不加锁情况下读取共享数据结构，通过在更新数据时延迟释放旧版本的数据来实现
    struct srcu_struct srcu;
    struct notifier_block __rcu *head;
};
```

4、notifier chain初始化

内核提供一套宏初始化各个类型的通知链（动态初始化）

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

静态初始化通知链（静态初始化）

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

#define SRCU_NOTIFIER_INIT(name, pcpu)              \
    {                           \
        .mutex = __MUTEX_INITIALIZER(name.mutex),   \
        .head = NULL,                   \
        .srcu = __SRCU_STRUCT_INIT(name.srcu, pcpu),    \
    }

#define ATOMIC_NOTIFIER_HEAD(name)              \
    struct atomic_notifier_head name =          \
        ATOMIC_NOTIFIER_INIT(name)
#define BLOCKING_NOTIFIER_HEAD(name)                \
    struct blocking_notifier_head name =            \
        BLOCKING_NOTIFIER_INIT(name)
#define RAW_NOTIFIER_HEAD(name)                 \
    struct raw_notifier_head name =             \
        RAW_NOTIFIER_INIT(name)
```

SRCU 通知链不能使用静态方法，内核提供了一个动态的初始化函数

```
/**
 *  srcu_init_notifier_head - Initialize an SRCU notifier head
 *  @nh: Pointer to head of the srcu notifier chain
 *
 *  Unlike other sorts of notifier heads, SRCU notifier heads require
 *  dynamic initialization.  Be sure to call this routine before
 *  calling any of the other SRCU notifier routines for this head.
 *
 *  If an SRCU notifier head is deallocated, it must first be cleaned
 *  up by calling srcu_cleanup_notifier_head().  Otherwise the head's
 *  per-cpu data (used by the SRCU mechanism) will leak.
 */
void srcu_init_notifier_head(struct srcu_notifier_head *nh)
{
    mutex_init(&nh->mutex);
    if (init_srcu_struct(&nh->srcu) < 0)
        BUG();
    nh->head = NULL;
}
```

5、注册 / 注销通知链

内核提供的最基本的注册通知链函数
通过判断 **priority** 的大小，倒序插入

```
/*
 *  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) {
        if (unlikely((*nl) == n)) {
            WARN(1, "double register detected");
            return 0;
        }
        if (n->priority > (*nl)->priority)
            break;
        nl = &((*nl)->next);
    }
    n->next = *nl;
    rcu_assign_pointer(*nl, n);
    return 0;
}
```

注销函数：先找到此节点，然后从链表中删除一个操作。因为插入 / 删除操作都是临界资源，需要使用 `rcu`机制保护起来。同理，内核通过包装核心的注册 / 注销函数，实现上述说的四种 `notifier chain`

```
extern int atomic_notifier_chain_register(struct atomic_notifier_head *nh,
        struct notifier_block *nb);
extern int blocking_notifier_chain_register(struct blocking_notifier_head *nh,
        struct notifier_block *nb);
extern int raw_notifier_chain_register(struct raw_notifier_head *nh,
        struct notifier_block *nb);
extern int srcu_notifier_chain_register(struct srcu_notifier_head *nh,
        struct notifier_block *nb);

extern int atomic_notifier_chain_unregister(struct atomic_notifier_head *nh,
        struct notifier_block *nb);
extern int blocking_notifier_chain_unregister(struct blocking_notifier_head
*nh,
        struct notifier_block *nb);
extern int raw_notifier_chain_unregister(struct raw_notifier_head *nh,
        struct notifier_block *nb);
extern int srcu_notifier_chain_unregister(struct srcu_notifier_head *nh,
        struct notifier_block *nb);
```

6、通知函数

当某种事件需要发生的时候，需要调用内核提供的通知函数 `notifier call` ，来通知注册过响应的时间子系统

```
/**
 * notifier_call_chain - Informs the registered notifiers about an event.
 *  @nl:        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:    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.
 *  @returns:   notifier_call_chain returns the value returned by the
 *          last notifier function called.
 */
static int 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) {
        next_nb = rcu_dereference_raw(nb->next);

#ifdef CONFIG_DEBUG_NOTIFIERS
        if (unlikely(!func_ptr_is_kernel_text(nb->notifier_call))) {
            WARN(1, "Invalid notifier called!");
            nb = next_nb;
            continue;
        }
#endif
        // 调用注册的回调函数
        ret = nb->notifier_call(nb, val, v);

        if (nr_calls)
            (*nr_calls)++;

        // 停止的mask就返回，否则继续
        if (ret & NOTIFY_STOP_MASK)
            break;
        nb = next_nb;
        nr_to_call--;
    }
    return ret;
}
NOKPROBE_SYMBOL(notifier_call_chain);
```

提供四个不同类型的通知函数

```
extern int atomic_notifier_call_chain(struct atomic_notifier_head *nh,
        unsigned long val, void *v);
extern int blocking_notifier_call_chain(struct blocking_notifier_head *nh,
        unsigned long val, void *v);
extern int raw_notifier_call_chain(struct raw_notifier_head *nh,
        unsigned long val, void *v);
extern int srcu_notifier_call_chain(struct srcu_notifier_head *nh,
        unsigned long val, void *v);
```

7、编程逻辑

（1）注册事件

- 静态初始化通知链的头部 BLOCKING_NOTIFIER_HEAD
- 定义回调函数 test_chain_notify (包裹在 struct notifier_block)
- 注册通知事件blocking_notifier_chain_register
- 注销通知事件blocking_notifier_chain_unregister

（2）触发事件

- 获取通知链的头部并触发事件 blocking_notifier_call_chain
- 打印错误信息

notifier.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/notifier.h>


BLOCKING_NOTIFIER_HEAD(test_chain_head);
EXPORT_SYMBOL_GPL(test_chain_head);

int register_test_notifier(struct notifier_block *nb)
{
    return blocking_notifier_chain_register(&test_chain_head, nb);
}

int unregister_test_notifier(struct  notifier_block *nb)
{
    return blocking_notifier_chain_unregister(&test_chain_head, nb);
}

static int test_chain_notify(struct notifier_block *nb,unsigned long mode,
void *_unused)
{
    printk(KERN_EMERG "notifier: test_chain_notify!\n");        //回调处理函数
    return 0;
}

static struct notifier_block test_chain_nb = {
    .notifier_call = test_chain_notify,
};


static int notifier_test_init(void)
{
    printk(KERN_EMERG "notifier: notifier_test_init!\n");
    register_test_notifier(&test_chain_nb);                          //注册
notifier事件

    return 0;
}
```

```
static void notifier_test_exit(void)
{
    printk(KERN_EMERG "notifier: notifier_test_exit!\n");
    unregister_test_notifier(&test_chain_nb);
}

module_init(notifier_test_init);
module_exit(notifier_test_exit);
MODULE_LICENSE("GPL v2");
```

call.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/notifier.h>

struct blocking_notifier_head test_chain_head;

static int call_notifier_call_chain(unsigned long val)
{
    int ret = blocking_notifier_call_chain(&test_chain_head, val, NULL);
    return notifier_to_errno(ret);
}

static int call_test_init(void)
{
    printk(KERN_EMERG "notifier: call_test_init!\n");
    call_notifier_call_chain(123); //在init函数中触发事件

    return 0;
}

static void call_test_exit(void)
{
    printk(KERN_EMERG "notifier: call_test_exit!\n");
}
module_init(call_test_init);
module_exit(call_test_exit);
MODULE_LICENSE("GPL v2");
```

Makefile

```
ifneq ($(KERNELRELEASE),)
obj-m :=notifiler.o
else
KDIR :=/lib/modules/$(shell uname -r)/build
all:
    make -C $(KDIR) M=$(PWD) modules
clean:
    rm -f *.ko *.o *.mod.o *.mod.c *.symvers *.order
endif
```