semaphore工作原理及其使用案例

一 semaphore工作原理

我们经常在kernel源码中声明semaphore,如下两种方式:

- 静态申请:

```
#define DEFINE_SEMAPHORE(name)  \
    struct semaphore name = __SEMAPHORE_INITIALIZER(name, 1)
```

- 动态申请

```
static inline void sema_init(struct semaphore *sem, int val)
{
    static struct lock_class_key __key;
    *sem = (struct semaphore) __SEMAPHORE_INITIALIZER(*sem, val);
    lockdep_init_map(&sem->lock.dep_map, "semaphore->lock", &__key, 0);
}
```

它们唯一的区别就是val的数值不同,静态申请的semaphore val=1,即semaphore只能只有一个线程使用,等待的线程task进入休眠状态.
而动态申请的semaphore,val是用户可以自己设定的.如果val > 1,则可以多个线程同时使用此semaphore,直到val <= 0. 其他使用此semaphore的线程task进入休眠等待状态.

1.1 主要结构体分析

主要涉及到两个结构体,分析如下:
semaphore本身的结构体变量:

```
/* Please don't access any members of this structure directly */
struct semaphore {
    raw_spinlock_t      lock;
    unsigned int        count;
    struct list_head    wait_list;
};
```

使用spin lock来保护成员变量count和wait_list.
semaphore等待者结构体,即需要某个相同semaphore的线程task都会填充下面这个结构体:

```
/* Functions for the contended case */

struct semaphore_waiter {
    struct list_head list;
    struct task_struct *task;
    bool up;
```

```
};
```

等待semaphore的线程task全部放入链表中,up参数是在释放semaphore的时候设置为true,下面会分析到.

总结上面两个结构体: semaphore_waiter成员变量list是保存需要获取某个semaphore的线程task信息, 这个某个semaphore是谁,怎么关联起来,就是通过结构体semaphore成员变量wait_list,即只要想获取某个semaphore而没有获取成功,则将此semaphore挂载到list链表头上.在释放semaphore的时候,会从这个链表中获取结构体sema_waiter实体,进而也获取到了task信息了.后面详细分析.

1.2 如何获取semaphore

一般在kernel source code里面都是down来获取semaphore的.有下面几种方式来获取想要的semaphore的:

```
void down(struct semaphore *sem)
int down_interruptible(struct semaphore *sem)
int down_killable(struct semaphore *sem)
int down_trylock(struct semaphore *sem)
int down_timeout(struct semaphore *sem, long timeout)
```

分别解释如下:

1. down最简单的一种,如果task获取此semaphore,则把此task置为休眠状态,直到semaphore释放,其实就是设置此task调度状态为:TASK_INTERRUPTIBLE
2. down_interruptible是down的一种变种,区别是获取semaphore失败的task可以进入可中断的休眠状态.设置此task的调度状态为:TASK_INTERRUPTIBLE
3. down_killable也是down的变种,如果没有获取semaphore的task进入休眠状态,可被致命信号中断.
4. down_trylock其实是直接判断结构体semaphore成员变量count的数值是否<0
5. down_timeout是down的另一个变种,是获取semaphore失败的task,进入休眠时间timeout之后还没有获取semaphore,则报error.

下面开始讲解上面的实现原理:

```
/**
 * down - acquire the semaphore
 * @sem: the semaphore to be acquired
 *
 * Acquires the semaphore.  If no more tasks are allowed to acquire the
 * semaphore, calling this function will put the task to sleep until the
 * semaphore is released.
 *
 * Use of this function is deprecated, please use down_interruptible() or
 * down_killable() instead.
 */
void down(struct semaphore *sem)
{
    unsigned long flags;

    raw_spin_lock_irqsave(&sem->lock, flags);
    if (likely(sem->count > 0))
        sem->count--;
    else
        __down(sem);
    raw_spin_unlock_irqrestore(&sem->lock, flags);
```

```c
}
EXPORT_SYMBOL(down);

/**
 * down_interruptible - acquire the semaphore unless interrupted
 * @sem: the semaphore to be acquired
 *
 * Attempts to acquire the semaphore.  If no more tasks are allowed to
 * acquire the semaphore, calling this function will put the task to sleep.
 * If the sleep is interrupted by a signal, this function will return -EINTR.
 * If the semaphore is successfully acquired, this function returns 0.
 */
int down_interruptible(struct semaphore *sem)
{
    unsigned long flags;
    int result = 0;

    raw_spin_lock_irqsave(&sem->lock, flags);
    if (likely(sem->count > 0))
        sem->count--;
    else
        result = __down_interruptible(sem);
    raw_spin_unlock_irqrestore(&sem->lock, flags);

    return result;
}
EXPORT_SYMBOL(down_interruptible);

/**
 * down_killable - acquire the semaphore unless killed
 * @sem: the semaphore to be acquired
 *
 * Attempts to acquire the semaphore.  If no more tasks are allowed to
 * acquire the semaphore, calling this function will put the task to sleep.
 * If the sleep is interrupted by a fatal signal, this function will return
 * -EINTR.  If the semaphore is successfully acquired, this function returns
 * 0.
 */
int down_killable(struct semaphore *sem)
{
    unsigned long flags;
    int result = 0;

    raw_spin_lock_irqsave(&sem->lock, flags);
    if (likely(sem->count > 0))
        sem->count--;
    else
        result = __down_killable(sem);
    raw_spin_unlock_irqrestore(&sem->lock, flags);

    return result;
}
EXPORT_SYMBOL(down_killable);
```

```c

/**
 * down_trylock - try to acquire the semaphore, without waiting
 * @sem: the semaphore to be acquired
 *
 * Try to acquire the semaphore atomically.  Returns 0 if the semaphore has
 * been acquired successfully or 1 if it it cannot be acquired.
 *
 * NOTE: This return value is inverted from both spin_trylock and
 * mutex_trylock!  Be careful about this when converting code.
 *
 * Unlike mutex_trylock, this function can be used from interrupt context,
 * and the semaphore can be released by any task or interrupt.
 */
int down_trylock(struct semaphore *sem)
{
    unsigned long flags;
    int count;

    raw_spin_lock_irqsave(&sem->lock, flags);
    count = sem->count - 1;
    if (likely(count >= 0))
        sem->count = count;
    raw_spin_unlock_irqrestore(&sem->lock, flags);

    return (count < 0);
}
EXPORT_SYMBOL(down_trylock);

/**
 * down_timeout - acquire the semaphore within a specified time
 * @sem: the semaphore to be acquired
 * @timeout: how long to wait before failing
 *
 * Attempts to acquire the semaphore.  If no more tasks are allowed to
 * acquire the semaphore, calling this function will put the task to sleep.
 * If the semaphore is not released within the specified number of jiffies,
 * this function returns -ETIME.  It returns 0 if the semaphore was acquired.
 */
int down_timeout(struct semaphore *sem, long timeout)
{
    unsigned long flags;
    int result = 0;

    raw_spin_lock_irqsave(&sem->lock, flags);
    if (likely(sem->count > 0))
        sem->count--;
    else
        result = __down_timeout(sem, timeout);
    raw_spin_unlock_irqrestore(&sem->lock, flags);

```

```
        return result;
    }
    EXPORT_SYMBOL(down_timeout);

    static noinline void __sched __down(struct semaphore *sem)
    {
        __down_common(sem, TASK_UNINTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
    }

    static noinline int __sched __down_interruptible(struct semaphore *sem)
    {
        return __down_common(sem, TASK_INTERRUPTIBLE, MAX_SCHEDULE_TIMEOUT);
    }

    static noinline int __sched __down_killable(struct semaphore *sem)
    {
        return __down_common(sem, TASK_KILLABLE, MAX_SCHEDULE_TIMEOUT);
    }

    static noinline int __sched __down_timeout(struct semaphore *sem, long timeout)
    {
        return __down_common(sem, TASK_UNINTERRUPTIBLE, timeout);
    }
```

通过上面的代码获取semaphore的时候,分两个处理逻辑:

1. 获取semaphore成功,直接对count减一操作
2. 获取semaphore失败,根据对获取失败semaphore进程的休眠状态分别处理

有必要非常注意的是,在整个的获取semaphore的过程中都被spin lock 保护起来了.使用raw_spin_lock_irqsave来关闭本地CPU的中断.

我们在来看它们的通用核心函数:__down_common:

```
/* Functions for the contended case */
/* 保存获取失败semaphore的进程信息 */
struct semaphore_waiter {
    struct list_head list;
    struct task_struct *task;
    bool up;
};

/*
 * Because this function is inlined, the 'state' parameter will be
 * constant, and thus optimised away by the compiler.  Likewise the
 * 'timeout' parameter for the cases without timeouts.
 */
static inline int __sched __down_common(struct semaphore *sem, long state,
                                long timeout)
{   /*获取semaphore失败的进程*/
    struct task_struct *task = current;
    struct semaphore_waiter waiter;
    /*填充等待semaphore进程信息结构体变量,失败的全部放在一个链表中*/
    list_add_tail(&waiter.list, &sem->wait_list);
    waiter.task = task;
    /*获取semaphore失败的进程,up变量设置为false,在是否semaphore的时候会置为true*/
    waiter.up = false;
```

```
    /*根据进程设定的状态信息执行不同的调度操作*/
    for (;;) {
        if (signal_pending_state(state, task))
            goto interrupted;
        if (unlikely(timeout <= 0))
            goto timed_out;
        /*设置进程状态为state*/
        __set_task_state(task, state);
        /*必须关闭spin lock,因为调用__down_common函数的时候,已经调用了
       raw_spin_lock_irqsave spin lock*/
        raw_spin_unlock_irq(&sem->lock);
        /*释放cpu,进入休眠状态.如果不释放spin lock而直接进入休眠状态,系统会panic*/
        timeout = schedule_timeout(timeout);
        raw_spin_lock_irq(&sem->lock);
        if (waiter.up)
            return 0;
    }
 /*timeout之后还没有获取semaphore,将此task waiter信息清空*/
 timed_out:
    list_del(&waiter.list);
    return -ETIME;
 /*休眠被中断唤醒,将此task的waiter信息从链表中清空*/
 interrupted:
    list_del(&waiter.list);
    return -EINTR;
}
```

1.3 如何释放semaphore

我们经常看到释放semaphore使用up函数,那么它的原理是什么的呢?

```
/**
 * up - release the semaphore
 * @sem: the semaphore to release
 *
 * Release the semaphore.  Unlike mutexes, up() may be called from any
 * context and even by tasks which have never called down().
 */
void up(struct semaphore *sem)
{
    unsigned long flags;

    raw_spin_lock_irqsave(&sem->lock, flags);
    /*如果waiter 链表为空,则将count++,即大部分情况下,count数值为1*/
    if (likely(list_empty(&sem->wait_list)))
        sem->count++;
    else
        __up(sem);/*获取waiter 链表成员,并唤醒此链表成员上的进程*/
    raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(up);

static noinline void __sched __up(struct semaphore *sem)
```

```
{    /*获取链表上第一个节点*/
    struct semaphore_waiter *waiter = list_first_entry(&sem->wait_list,
                        struct semaphore_waiter, list);
    /*将执行释放semaphore的链表元素从waiter链表中清除*/
    list_del(&waiter->list);
    waiter->up = true;   /*设置flag为true*/
    wake_up_process(waiter->task);   /*直接唤醒此进程,参与调度*/
}
```

从up函数的解释很有意思:
不像mutex, 释放semaphore,可以在任意的上下文,甚至没有调用down函数的进程都可以主动去释放semaphore.

整体比较简单

二 semaphore使用案例

下面写一个使用semaphore的例子如下:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/stat.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/device.h>
#include <linux/cdev.h>
#include <asm/uaccess.h>
#include <linux/delay.h>
#include <linux/semaphore.h>

static dev_t devid;
static struct class *cls = NULL;
static struct cdev mydev;
 /*静态和动态申请,都可以*/
//static DEFINE_SEMAPHORE(mysema);
static struct semaphore mysema;

static int my_open(struct inode *inode, struct file *file)
{
    int i;
    /*获取semaphore成功则,每隔1s打印一次log*/
    while(down_interruptible(&mysema) != 0);
    for(i = 0; i < 10; i++) {
        printk("semaphore test:%d\n", i);
        ssleep(1);
    }
    /*释放semaphore.后面每次open节点都会再次获取semaphore mysema*/
    up(&mysema);
    printk("open success!\n");
    return 0;
}

static int my_release(struct inode *inode, struct file *file)
{
```

```c
    printk("close success!\n");
    return 0;
}

//定义文件操作
static struct file_operations myfops = {
    .owner = THIS_MODULE,
    .open = my_open,
    .release = my_release,
};

static void hello_cleanup(void)
{
    cdev_del(&mydev);
    device_destroy(cls, devid);
    class_destroy(cls);
    unregister_chrdev_region(devid, 1);
}

static __init int hello_init(void)
{
    int result;
    /*动态创建一个semaphore mysema,并且count设置为1*/
    sema_init(&mysema, 1);

    //动态注册设备号
    if(( result = alloc_chrdev_region(&devid, 0, 1, "samar-alloc-dev") ) != 0) {
        printk("register dev id error:%d\n", result);
        goto err;
    } else {
        printk("register dev id success!\n");
    }
    //动态创建设备节点
    cls = class_create(THIS_MODULE, "samar-class");
    if(IS_ERR(cls)) {
        printk("create class error!\n");
        goto err;
    }

    if(device_create(cls, NULL, devid, "", "hello%d", 0) == NULL) {
        printk("create device error!\n");
        goto err;
    }
    //字符设备注册
    mydev.owner = THIS_MODULE;      //必要的成员初始化
    mydev.ops = &myfops;
    cdev_init(&mydev, &myfops);
    //添加一个设备
    result = cdev_add(&mydev, devid, 1);
    if(result != 0) {
        printk("add cdev error!\n");
        goto err;
    }
```

```
    printk(KERN_ALERT "hello init success!\n");
    return 0;
err:
    hello_cleanup();
    return -1;
}

static __exit void hello_exit(void)
{
    hello_cleanup();
    printk("helloworld exit!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("samarxie");
MODULE_DESCRIPTION("For semaphore use method");
```

semaphore原理比较容易.关键还是使用了spin lock
通过用户空间操作/dev/hello0节点:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int
main(void)
{
        int fd;
        char buf[100];
        int size;

        fd = open("/dev/hello0", O_RDWR);
        if(!fd) {
                perror("open");
                exit(-1);
        }
        printf("open success!\n");

        close(fd);
        return 0;
}
```