

Лекция 2.3 Потоки ядра и методы синхронизации

Разработал: Максимов А.Н.

Версия 1.7

Содержане лекции

- SoftIRQ
- Tasklets
- Поток ядра
- Методы синхронизации в ядре
- Прием, передача информации

Softirq

Есть несколько видов:

HI_SOFTIRQ, TIMER_SOFTIRQ, NET_TX_SOFTIRQ, NET_RX__SOFTIRQ,
SCSI_SOFTIRQ, TASKLET_SOFTIRQ

Структура данных softirq_action

action, data

Возможны следующие операции:

Инициализация open_softirq

Активация rise_softirq

Выполнение do_softirq

Если irq_exit, ksoftirqd

Свойства Softirq

- Число softirq фиксированное и определяется при компиляции (см. `Linux/include/linux/interrupt.h`)
- Softirqs запускаются в порядке приоритета.
- Softirqs привязаны к ядрам CPU.
- Гарантируется, что softirq будет запущено на том ядре на котором оно запланировано к выполнению.
- Выполняются в контексте прерывания и имеют все его ограничения
- Не могут быть выполнены
- Выполняются атомарно
- На разных ядрах могут быть запущены несколько softirq даже одного тип

Как происходит запуск Softirq

Подсистема softirq включает несколько потоков ядра, по одному потоку на ядро (ksoftirqd/0, ksoftirqd/1, ksoftirqd/2 ...), которые запускаю функции обработчики softirq для различных softirq.

Потоки для softirq стартуют при начальной инициализации ядра (`kernel/softirq.c`):

```
static __init int spawn_ksoftirqd(void)
{
    register_cpu_notifier(&cpu_nfb);

    BUG_ON(smpboot_register_percpu_thread(&softirq_threads));
    return 0;
}

early_initcall(spawn_ksoftirqd);
```

Как происходит запуск Softirq

Softirq вызываются в нескольких местах ядра, наиболее распространенное — после обработчика прерывания (могут перепланировать сами себя)

do_IRQ() (Вызов верхней части обработчика прерывания, который маскирует все прерывания и вызывает)

| irq_exit() (Размаскирование прерываний)

| invoke_softirq() (Ядро проверяет про существующие в наличии softirq)

| do_softirq() (Выполнение softirq (верхней части обработчика прерываний))

Тут можно прочитать про распределение net_softirq между ядрами:

<http://natsys-lab.blogspot.ru/2012/09/linux-scaling-softirq-among-many-cpu.html>

Замечания по работе с Tasklet-ами

Тасклеты обычно вызываются в обработчике прерывания для выполнения длительных задач.

Тасклеты выполняются в контексте прерывания (не могут спать, не могут обращаться к памяти пользовательских задач)

Тасклеты могут пробуждать пользовательские процессы.

Tasklet

Для представления таксклета используется структура `struct tasklet_struct`.

Определение таксклета:

```
DECLARE_TASKLET (module_tasklet, /* name */  
                 module_do_tasklet, /* function */  
                 0 /* data */  
);
```

Запланировать таксклет для выполнения (interrupt handler):

```
tasklet_schedule(&module_tasklet);
```

Есть `tasklet_hi_schedule` function для определены высокоприоритетных таксклетов.

Обычно, таксклеты выполняются сразу после hard irqs

Один и тот же таксклет не может выполняться на разных процессорах одновременно.

Разные таксклеты могут выполняться параллельно.

tasklet_struct

Структура struct tasklet_struct имеет следующий формат:

```
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};
```

Определена в include/linux/interrupt.h

Пример работы с Tasklet-ами

```
struct tasklet_struct  my_tasklet;  
struct mydata { : } my_tasklet_data;  
void my_function ( unsigned long );
```

```
tasklet_init( &my_tasklet, &my_function, (unsigned long)&my_tasklet_data );  
tasklet_schedule( &my_tasklet );  
tasklet_kill( &my_tasklet );
```

Функции для работы с таксклетами определены в <linux/interrupt.h>

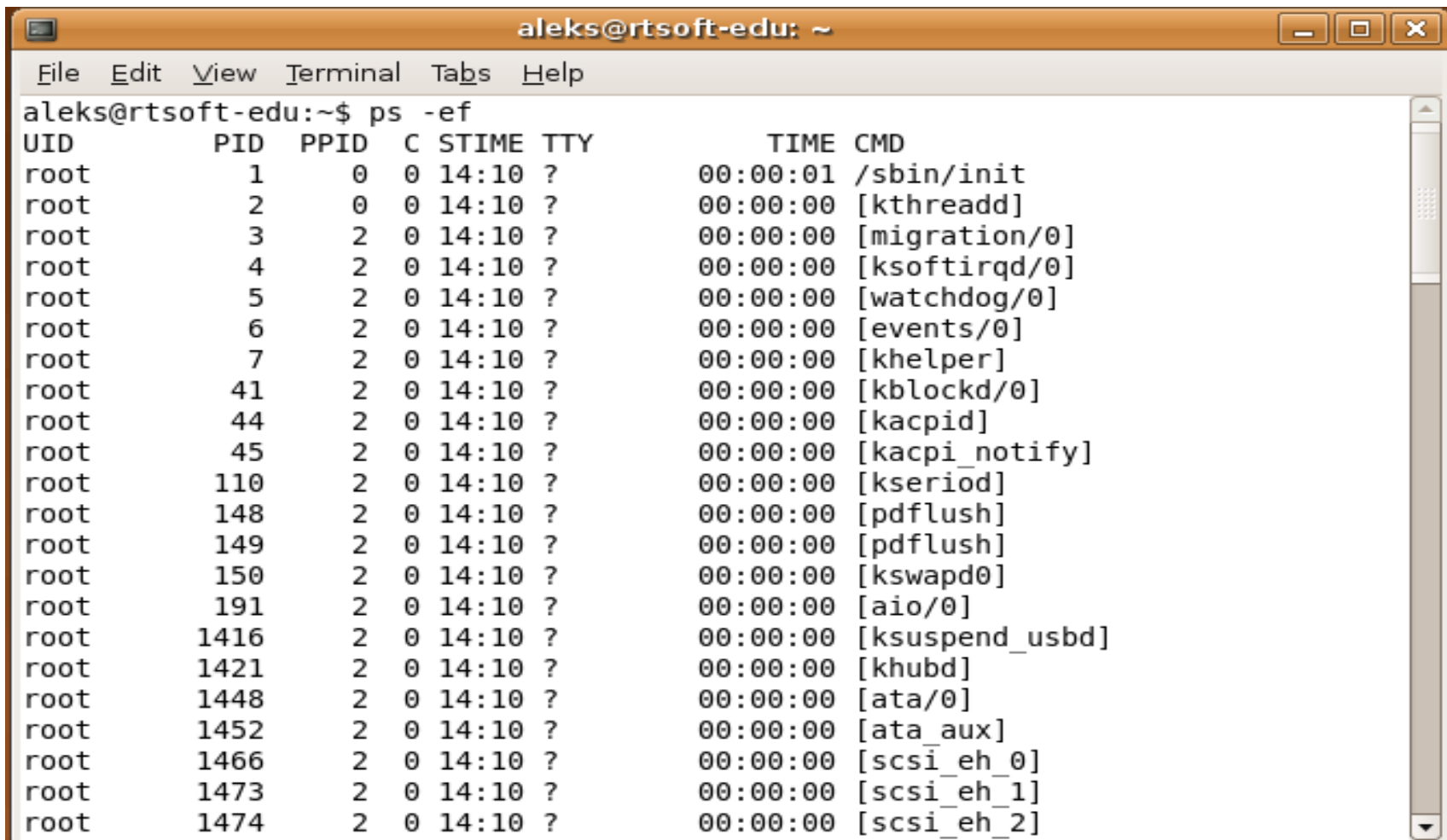
Пример работы с тасклетами и обработчиком прерывания можно найти тут:
<http://www.tune2wizard.com/kernel-programming-interrupts-and-tasklets/>

Потоки ядра

Назначение потоков ядра:

- Для обработки асинхронных событий
- Необходимо иметь доступ к структурам ядра
- Необходимо вызвать программу пользовательского уровня

Пример



A terminal window titled 'aleks@rtsoft-edu: ~' showing the output of the command 'ps -ef'. The window has a menu bar with 'File', 'Edit', 'View', 'Terminal', 'Tabs', and 'Help'. The output is a table of process information.

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	14:10	?	00:00:01	/sbin/init
root	2	0	0	14:10	?	00:00:00	[kthreadd]
root	3	2	0	14:10	?	00:00:00	[migration/0]
root	4	2	0	14:10	?	00:00:00	[ksoftirqd/0]
root	5	2	0	14:10	?	00:00:00	[watchdog/0]
root	6	2	0	14:10	?	00:00:00	[events/0]
root	7	2	0	14:10	?	00:00:00	[khelper]
root	41	2	0	14:10	?	00:00:00	[kblockd/0]
root	44	2	0	14:10	?	00:00:00	[kacpid]
root	45	2	0	14:10	?	00:00:00	[kacpi_notify]
root	110	2	0	14:10	?	00:00:00	[kseriod]
root	148	2	0	14:10	?	00:00:00	[pdflush]
root	149	2	0	14:10	?	00:00:00	[pdflush]
root	150	2	0	14:10	?	00:00:00	[kswapd0]
root	191	2	0	14:10	?	00:00:00	[aio/0]
root	1416	2	0	14:10	?	00:00:00	[ksuspend_usbd]
root	1421	2	0	14:10	?	00:00:00	[khubd]
root	1448	2	0	14:10	?	00:00:00	[ata/0]
root	1452	2	0	14:10	?	00:00:00	[ata_aux]
root	1466	2	0	14:10	?	00:00:00	[scsi_eh_0]
root	1473	2	0	14:10	?	00:00:00	[scsi_eh_1]
root	1474	2	0	14:10	?	00:00:00	[scsi_eh_2]

Создание потока ядра

Создание:

```
ret = kernel_thread  
    (mykthread,NULL,CLONE_FS|CLONE_FILES,CLONE_SIGHAND,SI  
    GCHLD)
```

```
static int mykthread(void *unused)  
{  
    .....  
    .....  
}
```

Функции для работы с потоками ядра

Создать:

```
struct task_struct *kthread_create(int (*threadfn)(void *data), void  
    *data, const char namefmt[], ...)
```

Создать и запустить:

```
struct task_struct *kthread_run(int (*threadfn)(void *data), void  
    *data, const char *namefmt, ...);
```

Потока ядра происходит может быть остановлен из вне:

```
int kthread_stop(struct task_struct *thread);
```

Поток ядра может быть запущен на указанном процессоре:

```
void kthread_bind(struct task_struct *k, unsigned int cpu);
```

Функции для работы с потоками ядра определены в `include/linux/kthread.h`

Пример работы с потоками ядра

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/wait.h>
#include <linux/kthread.h>

struct task_struct *ts;

int thread(void *data) {
    while(1) {
        printk("Hello. I am kernel thread! \n");    msleep(100);
        if (kthread_should_stop())    break;
    }
    return 0;
}

int init_module(void) {
    printk(KERN_INFO "init_module() called\n");
    ts=kthread_run(thread,NULL,"foo kthread");
    return 0;
}

void cleanup_module(void) {
    printk(KERN_INFO "cleanup_module() called\n");
    kthread_stop(ts);
}
```

printk_ratelimited

Для контроля частоты вывода можно использовать

```
printk_ratelimited(fmt, ...)
```

и

```
printk_once(...)
```

см. `linux\printk.h`

Часть 2: Методы синхронизации в ядре

- Атомарные переменные
- *Атомарные битовые поля*
- spinlock
- *Циклические блокировки чтения/записи*
- *Неблокирующая синхронизация (RCU)*
- семафоры
- mutex

Атомарные переменные

Атомарные операторы идеальны для ситуаций, в которых защищаемые данные просты, например, как счетчик.

Для представления атомарных переменных используется тип `atomic_t`.

Операции для работы с атомарными переменными описаны в

`/linux/include/asm-
<arch>/atomic.h`

Инициализация атомарных переменных производится при помощи макроса `ATOMIC_INIT`

Пример.

```
atomic_t my_cnt ATOMIC_INIT(0);  
atomic_set( &my_cnt, 0 );  
val = atomic_read( &my_counter );
```

Арифметические функции для работы с атомарными переменными:

```
atomic_add( 1, &my_cnt);  
atomic_inc( &my_cnt);  
atomic_sub( 1, &my_cnt);  
atomic_dec( &my_cnt);
```

Атомарные проверка/изменение

Атомарные операции проверка/изменение:

```
if (atomic_sub_and_test( 1, &my_cnt )) {  
    // my_counter равен нулю  
}  
  
if (atomic_dec_and_test( &my_cnt )) {  
    // my_counter равен нулю  
}  
  
if (atomic_inc_and_test( &my_cnt )) {  
    // my_counter равен нулю  
}  
  
if (atomic_add_negative( 1, &my_cnt)) {  
    // my_counter меньше нуля  
}  
  
val = atomic_add_return( 1, &my_cnt);  
val = atomic_sub_return( 1, &my_cnt);
```

СПИНЛОК

Обеспечивает, чтобы только один поток входил в критическую секцию. Второй поток будет крутиться в спинлоке до тех пор пока первый не выйдет:

```
#include <linux/spinlock.h>

spinlock_t mylock = SPIN_LOCK_UNLOCKED; /* Initialize */

/* Acquire the spinlock. This is inexpensive if there
 * is no one inside the critical section. In the face of
 * contention, spinlock() has to busy-wait.
 */

spin_lock(&mylock);

/* ... Critical Section code ... */

spin_unlock(&mylock); /* Release the lock */
```

<https://www.kernel.org/doc/Documentation/locking/spinlocks.txt>

Спинлок реализация arm

Спинлоки используются в коде в котором нельзя засыпать?
например, обработчики прерываний. Реализация
спинлоков — специфична для архитектуры и реализуется
на асемблере. Для ARM

arch/arm/include/asm/spinlock.h":

```
static inline void arch_spin_lock(arch_spinlock_t *lock)
```

```
static inline void arch_spin_unlock(arch_spinlock_t *lock)
```

Нужные инструкции ассемблера

LDREX: Load Register Exclusive

Optimistically loads a value from memory into a register assuming that nothing else will change the value in memory while we are working on it, which makes the spin lock mechanism possible.

STREX: Store Register Exclusive

STREXEQ: STREX with optional condition code EQ

TEQ: Test For Equality of Two 32-bit Value

TEQEQ: TEQ with the optional condition code EQ

BNE: Branch if Negative

STR: Store Register to Memory

Дополнительно:

1. <http://thinkingeek.com/2013/01/09/arm-assembler-raspberry-pi-chapter-1/>
2. http://infocenter.arm.com/help/topic/com.arm.doc.dui0489f/DUI0489F_arm_assembler_reference.pdf

arch_spin_lock

arch_spin_lock()

```
static inline void arch_spin_lock(arch_spinlock_t *lock)
{
    unsigned long tmp;
    __asm__ __volatile__(
"1: ldrex  %0, [%1]\n" /* exclusively load lock into %0, i.e. tmp */
"   teq %0, #0\n"      /* test equality against value 0 */
    WFE("ne")          /* special care for Thumb-2 WFE instructions */
"   strex %0, %2, [%1]\n" /* exclusively store 1 into %0 of lock */
"   teqeq %0, #0\n" /* test equality of %0, i.e. tmp against 0 value */
"   bne 1b"            /* branch on negative to 1 to try again */
    : "=&r" (tmp) /* tmp: output, referred to by %0; r: use register to store */
    : "r" (&lock->lock), "r" (1) /* inputs, %1, %2 */
    : "cc"); /* clobbered register cc (condition code) will be modified */
    smp_mb(); /* SMP memory barrier to protect the &lock->lock */
}
```

arch_spin_unlock

arch_spin_unlock()

```
static inline void arch_spin_unlock(arch_spinlock_t *lock)
{
    smp_mb();                /* SMP memory barrier */
    __asm__ __volatile__(
"    str %1, [%0]\n"        /* store 0 into &lock->lock, i.e. unlock */
:                            /* output: none */
: "r" (&lock->lock), "r" (0) /* input %0, %1 */
: "cc");                  /* clobbered register cc (condition code) will be set/modified */
    dsb_sev();              /* Data Synchronization Barrier */
}
```


arch_spin_unlock

Что происходит:

читаем &lock->lock в регистр

Если регистр равен 0 мы захватываем лок, срабатываем барьер и мы продолжаем выполнение

Если регистр равен 1, мы возвращаемся обратно в цикл сравнения ("busy looping")

"цикл проверок" до тех пор пока не захватим лок и продолжим дальше

spin_lock реализация x86

<http://sklinuxblog.blogspot.ru/2015/04/linux-spinlock-implementation-on-x8664.html>

Пример использования спинлоков

```
#include <linux/module.h>
#include <linux/timer.h>
#include <linux/spinlock.h>

struct global_data { /* A global struct acting as critical section */
    spinlock_t lock; /* Spinlock */
    int count; /* Counter */
};

struct global_data kt_global_data;
struct timer_list kt_timer; /* A timer list */
void timer_callback(unsigned long arg) /* Timer callback */
{
    printk(KERN_INFO "Entering: %s\n", __FUNCTION__);
    struct global_data *data = (struct global_data *)arg;
    spin_lock(&(data->lock));
    data->count++;
    spin_unlock(&(data->lock));
    mod_timer(&kt_timer, jiffies + 10*HZ); /* restarting timer */
}
```

<https://davejingtian.org/2014/07/09/kt-use-kernel-timers-in-the-linux-kernel/>

Пример использования спинлоков

```
static void kt_init_timer(void) /* Init the timer */
{
    init_timer(&kt_timer);
    kt_timer.function = timer_callback;
    kt_timer.data = (unsigned long)(&kt_global_data);
    kt_timer.expires = jiffies + 10*HZ; /* 10 second */
    add_timer(&kt_timer); /* Starting the timer */
    printk(KERN_INFO "kt_timer is started\n");
}

static void kt_do_the_work(void) /* The normal module worker */
{
    printk(KERN_INFO "Before %s, count = %d\n", __FUNCTION__, kt_global_data.count);
    spin_lock(&(kt_global_data.lock));
    kt_global_data.count++;
    spin_unlock(&(kt_global_data.lock));
    printk(KERN_INFO "After %s, count = %d\n", __FUNCTION__, kt_global_data.count);
}
```

Пример использования спинлоков

```
static int __init kt_init(void)
{
    printk(KERN_INFO "Entering: %s\n", __FUNCTION__);
    memset(&kt_global_data, 0x0, sizeof(struct global_data)); /* Init the global data */
    spin_lock_init(&(kt_global_data.lock)); /* Init the spinlock */
    kt_init_timer(); /* Init the timer */
    kt_do_the_work(); /* Do our job */
    return 0;
}

static void __exit kt_exit(void)
{
    printk(KERN_INFO "exiting kt module\n");
    del_timer_sync(&kt_timer); /* Delete the timer */
    printk(KERN_INFO "kt_global_data.count = [%d]\n", kt_global_data.count);
}
```

mutex

- Помещает поток в сон
- Предпочтительно использовать, когда время ожидания больше 2 переключений контекста
- Дополнительные правила:
- - если в критической секции могут быть schedule, preempt, sleep on wait queue (to mutex)
- - если в обработчике прерывания, то spinlock

Пример использования

```
#include <linux/mutex.h>
```

```
/* Statically declare a mutex. To dynamically  
create a mutex, use mutex_init() */
```

```
static DEFINE_MUTEX(mymutex);
```

```
/* Acquire the mutex. This is inexpensive if there  
* is no one inside the critical section. In the face of  
* contention, mutex_lock() puts the calling thread to sleep.  
*/
```

```
mutex_lock(&mymutex);
```

```
/* ... Critical Section code ... */
```

```
mutex_unlock(&mymutex); /* Release the mutex */
```

Рекомендации по использованию

	IRQ Handler A	IRQ Handler B	Softirq A	Softirq B	Tasklet A	Tasklet B	Timer A	Timer B	User Context A	User Context B
IRQ Handler A	None									
IRQ Handler B	SLIS	None								
Softirq A	SLI	SLI	SL							
Softirq B	SLI	SLI	SL	SL						
Tasklet A	SLI	SLI	SL	SL	None					
Tasklet B	SLI	SLI	SL	SL	SL	None				
Timer A	SLI	SLI	SL	SL	SL	SL	None			
Timer B	SLI	SLI	SL	SL	SL	SL	SL	None		
User Context A	SLI	SLI	SLBH	SLBH	SLBH	SLBH	SLBH	SLBH	None	
User Context B	SLI	SLI	SLBH	SLBH	SLBH	SLBH	SLBH	SLBH	MLI	None

SLIS	spin_lock_irqsave
SLI	spin_lock_irq
SL	spin_lock
SLBH	spin_lock_bh
MLI	mutex_lock_interruptible

https://mchehab.fedorapeople.org/kernel_docs_latex/kernel-hacking.pdf

Литература

1. <http://www.cs.columbia.edu/~junfeng/10sp-w4118/lectures/l11-synch-linux.pdf>
2. A. Rubini Linux Device Drivers.
3. <http://www.cs.utexas.edu/~witchel/372/lectures/12.KernelSync.pdf>
4. М. Тим Джонс, Анатомия методов синхронизации Linux
<http://www.ibm.com/developerworks/ru/library/l-linux-synchronization/>
5. <http://www.slideshare.net/susantsahani/synchronization-linux>
6. <http://kukuruku.co/hub/nix/multitasking-in-the-linux-kernel-interrupts-and-tasklets>
7. https://wr.informatik.uni-hamburg.de/media/teaching/wintersemester_2014_2015/pk1415-concurrency.pdf
8. <http://people.cs.pitt.edu/~ouyang/20150225-kernel-concurrency.html>
9. <http://www.tune2wizard.com/kernel-programming-kernel-threads-synchronization-techniques-workqueue/>

Спинлоки и дедлоки. Тут dedlock

```
void my_func(.....) {  
    spin_lock(&mylock);  
  
    .....  
    spin_unlock(&mylock);  
}  
  
int my_intr(.....) {  
    spin_lock(&mylock);  
  
    .....  
    spin_unlock(&mylock);  
  
    .....  
}
```

<http://www.geeksofpune.in/files/kerneldebugging-2.pdf>

Спинлоки и дедлоки. Тут dedlock

my_func(.....)

- spin_lock(&mylock);
- Hardware interrupt (irq) occurs
- do_IRQ
 - my_intr
 - spin_lock(&mylock); ---> DEADLOCK

<http://www.geeksofpune.in/files/kerneldebugging-2.pdf>

Спинлоки и дедлоки. Так правильно

```
void my_func(.....) {  
    spin_lock_irqsave(&mylock, flags);  
    .....  
    spin_unlock_irqrestore(&mylock, flags);  
}  
  
int my_intr(.....) {  
    spin_lock(&mylock);  
    .....  
    spin_unlock(&mylock);  
}
```

<http://www.geeksofpune.in/files/kerneldebugging-2.pdf>

Спинлоки и дедлоки. Так правильно

```
void my_func(.....) {
```

```
    spin_lock(&mylock);
```

```
    .....
```

```
    spin_unlock(&mylock);
```

```
}
```

```
int my_intr(.....) {
```

```
    if (!spin_trylock(&mylock))
```

```
    { .... return;}
```

```
    .....
```

```
    spin_unlock(&mylock);
```

```
    .....
```

```
}
```

<http://www.geeksofpune.in/files/kerneldebugging-2.pdf>

Блокировки чтения/записи.

В большинстве случаев доступ к данным характеризуется большим числом читающих процессов и меньшим числом пишущих (доступ к данным для чтения более распространен, чем доступ для записи). Для поддержки такой модели были созданы взаимные блокировки чтения/записи.

```
rwlock_t my_rwlock;
```

```
rwlock_init( &my_rwlock );
```

```
write_lock( &my_rwlock ); // критическая секция -- разрешено чтение и запись
```

```
....
```

```
write_unlock( &my_rwlock );
```

```
read_lock( &my_rwlock ); // критическая секция -- разрешено только чтение
```

```
....
```

```
read_unlock( &my_rwlock );
```

Пример использования блокировки чтения/записи.

```
struct el {
    struct list_head list;
    long key;
    spinlock_t mutex;
    int data;
};

rwlock_t listmutex;
struct el head;
int search(long key, int *result) {
    struct list_head *lp; struct el *p;
    read_lock(&listmutex);
    list_for_each_entry(p, head, lp) {
        if (p->key == key) {
            *result = p->data;
            read_unlock(&listmutex);
            return 1;
        }
    }
    read_unlock(&listmutex);
    return 0;
}
```

Пример использования блокировки чтения/записи.

```
int delete(long key)    {
    struct el *p;

    write_lock(&listmutex);
    list_for_each_entry(p, head, lp) {
        if (p->key == key) {
            list_del(&p->list);
            write_unlock(&listmutex);

            kfree(p);
            return 1;
        }
    }
    write_unlock(&listmutex);
    return 0;
}
```


RCU API

RCU API включает следующие основные функции:

```
rcu_read_lock()  
rcu_read_unlock()  
synchronize_rcu() / call_rcu()  
rcu_assign_pointer()  
rcu_dereference()
```

Дополнительно можно прочесть:

<http://www2.rdrop.com/users/paulmck/RCU/RCU.TU-Dresden.2012.05.15a.pdf>

<http://lwn.net/Articles/262464/>

<https://www.cse.iitb.ac.in/~puru/courses/spring15/cs401/exercises.html>

www.cs.pdx.edu/~walpole/class/cs510/spring2014/slides/12.pptx