内核同步

张雷

南京大学计算机系

2018-10-16

Kernel Synchronization

Synchronization

自旋锁: Spin lock

信号量: Semaphore

互斥锁: Mutex

读写锁: Read write lock

RCU: Read-Copy-Update

Lab3

Kernel Synchronization

- Synchronization
- 自旋锁: Spin lock
- 信号量: Semaphore
- 互斥锁: Mutex
- 读写锁: Read write lock
- RCU: Read-Copy-Update
- Lab3

Synchronization

Processes can execute concurrently, concurrent access to shared data may result in data inconsistency, require mechanisms to ensure the orderly execution of cooperating processes
Example:

```
while (true) {
    /* produce an item */

    while(counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
while (true) {
    while(counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item */

}
```

Race condition: a heisenbug

`counter++` could be implemented as

- register1 = counter
- register1 = register1 + 1
- counter = register1

`counter--` could be implemented as

- register2 = counter
- register2 = register2 - 1
- counter = register2

Consider this execution interleaving with "count = 5" initially:

- S0: producer execute register1 = counter register1 = 5
- S1: producer execute register1 = register1 + 1 register1 = 6
- S2: consumer execute register2 = counter register2 = 5
- S3: consumer execute register2 = register2 – 1 register2 = 4
- S4: producer execute counter = register1 counter = 6
- S5: consumer execute counter = register2 counter = 4

Race condition

Critical Section

code paths that access and manipulate shared data

Race Condition

The threads run a race between their operations, and the results of the program execution depends on who wins the race (order of access).

Solution to Critical-Section Problem

- Mutual exclusion
- Forward progress
- Bounded waiting

atomic operations

atomic counter

- atomic_t type defined in <linux/types.h>
- cpu guaranteed atomic ops: read/write a byte, aligned word...
- reference counter, prevent garbage values

```
atomic_t v;
atomic_t u = ATOMIC_INIT(0);

atomic_set(&v, 4);
atomic_add(2, &v);
atomic_inc(&v);
/* will print "7" */
printk("%d\n", atomic_read(&v));
```

```
typedef struct {
    int counter;
} atomic_t;

#define ATOMIC_INIT(i)  \
        { (i) }
#define atomic_set(v, i)    \
        ((v)->counter = (i))
#define atomic_read(v)    \
        ((v)->counter)
```

atomic counter operations

<asm/atomic.h>: atomically read/set/add/sub/inc/dec

```
ATOMIC_INIT(int i)        At declaration, initialize to i;

int atomic_read(atomic_t *v)          read the value of v; unnecessary;
void atomic_set(atomic_t *v, int i)  set v equal to i; unnecessary;
void atomic_add(int i, atomic_t *v)  add i to v;
void atomic_sub(int i, atomic_t *v)  subtract i from v;
void atomic_inc(atomic_t *v)          add one to v;
void atomic_dec(atomic_t *v)          subtract one from v;

int atomic_add_negative(int i, atomic_t *v)  return true if the result<0;

int atomic_add_return(int i, atomic_t *v)    return the new counter value;
int atomic_sub_return(int i, atomic_t *v)    return the new counter value;
int atomic_inc_return(int i, atomic_t *v)    return the new counter value;
int atomic_dec_return(int i, atomic_t *v)    return the new counter value;

int atomic_sub_and_test(int i, atomic_t *v) return true if the result is 0;
int atomic_dec_and_test(atomic_t *v)    return true if the result is 0;
int atomic_inc_and_test(atomic_t *v)    return true if the result is 0;
```

atomic bitwise operations

<asm/bitops.h>: lock prefix instructions

```
unsigned long word = 0;

set_bit(0, &word); /* bit zero is now set (atomically) */
set_bit(1, &word); /* bit one is now set (atomically) */
printk("%ul\n", word); /* will print "3" */
clear_bit(1, &word); /* bit one is now unset (atomically) */
change_bit(0, &word); /* bit zero is flipped; */

/* atomically sets bit zero and returns the previous value 0 */
if (test_and_set_bit(0, &word)) {
    /* never true ... */
}

/* legal; can mix atomic bit op with normal C */
word = 7;
```

atomic bitwise operations: API

atomically set/clear/change/return the nr-th bit starting from addr.

```
void set_bit(int nr, void *addr);
void clear_bit(int nr, void *addr);
void change_bit(int nr, void *addr);

int test_and_set_bit(int nr, void *addr);
int test_and_clear_bit(int nr, void *addr);
int test_and_change_bit(int nr, void *addr);

int test_bit(int nr, void *addr);
```

Kernel Synchronization

- Synchronization
- 自旋锁: Spin lock
- 信号量: Semaphore
- 互斥锁: Mutex
- 读写锁: Read write lock
- RCU: Read-Copy-Update
- Lab3

Synchronization: locks

Critical sections can span many functions. We need more than atomic integer/bitwise operations. A more general method of synchronization: locks

```
lock(l);            /*acquire(l) wait(l)*/
read_update_data();/*critical section*/
unlock(l);          /*release(l) signal(l)*/
```

How to implement a lock?

```
struct lock {
}

void acquire (lock *x) {
    disable interrupts;
}

void release (lock *x) {
    enable interrupts;
}
```

```
struct lock {
    int held = 0;
}
void acquire (lock *x) {
    while (x->held);
    x->held = 1;
}
void release (lock *x) {
    x->held = 0;
}
```

Synchronization: locks

Locking by disabling interrupts works for a single CPU

- disabling interrupts also disables kernel preemption: no clock/timers
- disable interrupts: do not interrupt me on this CPU. with few CPUs this will fail.

Solution: use hardware supported special instructions

`TEST_AND_SET`

If test is not true then set to be true, and return old value.
Atomically across all CPUs.

Now we have a spinlock: spin = busy waiting.

```
void spin_lock(lock)
{
    while(test_and_set(lock) == 1)
        ;
}
```

```
void spin_unlock(lock)
{
    lock=0;

}
```

Spinlock in Linux

`TEST_AND_SET` is expensive

- locking the bus, then test/check and set.

The Linux kernel use a different spinlock implementation: ticket spinlock.

- use two ticket: head, tail.
- less latency, more fair.

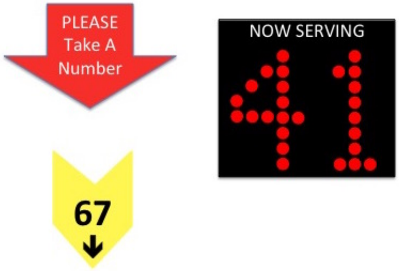


Figure 1: Tick lock concept

Linux spinlock implementation

```
<linux/spinlock_types.h>
typedef struct spinlock {
        union {
              struct raw_spinlock rlock;
        };
} spinlock_t;
typedef struct raw_spinlock {
        arch_spinlock_t raw_lock;
#ifdef CONFIG_GENERIC_LOCKBREAK
        unsigned int break_lock;
#endif
} raw_spinlock_t;
typedef struct arch_spinlock {
         union {
                __ticketpair_t head_tail;
                struct __raw_tickets {
                        __ticket_t head, tail;
                } tickets;
        };
} arch_spinlock_t;
```

Linux spinlock implementation

```c
static __always_inline void arch_spin_lock(arch_spinlock_t *lock)
{
        register struct __raw_tickets inc = { .tail = TICKET_LOCK_INC };

        inc = xadd(&lock->tickets, inc); /*increase tail*/

        if (likely(inc.head == inc.tail)) /*got lock*/
                goto out;

        for (;;) {
                unsigned count = SPIN_THRESHOLD;

                do {
                        inc.head = READ_ONCE(lock->tickets.head);
                        if (__tickets_equal(inc.head, inc.tail)) /*got lock*/
                                goto clear_slowpath;
                         cpu_relax(); /*just NOP instruction*/
                } while (--count);
                __ticket_lock_spinning(lock, inc.tail);
         }
clear_slowpath:
        __ticket_check_and_clear_slowpath(lock, inc.head);
out:
        barrier(); /*compiler will not change the order of operations*/
}
```

Linux spinlock implementation

spin_unlock(): arch/x86/include/asm/spinlock.h

```
static __always_inline void arch_spin_unlock(arch_spinlock_t *lock)
{
    if (TICKET_SLOWPATH_FLAG &&
        static_key_false(&paravirt_ticketlocks_enabled)) {
        __ticket_t head;

        BUILD_BUG_ON(((__ticket_t)NR_CPUS) != NR_CPUS);

        /*increase lock->tickets.head*/
        head = xadd(&lock->tickets.head, TICKET_LOCK_INC);

        if (unlikely(head & TICKET_SLOWPATH_FLAG)) {
            head &= ~TICKET_SLOWPATH_FLAG;
            __ticket_unlock_kick(lock, (head + TICKET_LOCK_INC));
        }
    } else
        __add(&lock->tickets.head, TICKET_LOCK_INC, UNLOCK_LOCK_PREFIX);
        /*increase lock->tickets.head*/
}
```

Spinlock API

<linux/spinlock.h>

```
DEFINE_SPINLOCK(mr_lock);
/*Dynamically initializes given spinlock_t*/
spin_lock_init(spinlock_t *lock);

void spin_lock(spinlock_t *lock);
/*Disables local interrupts and acquires given lock*/
void spin_lock_irq(spinlock_t *lock);
/* Save current state of local interrupts, disables local interrupts, and lock*
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);

void spin_unlock(spinlock_t *lock);
/*Releases given lock and enables local interrupts*/
void spin_unlock_irq(spinlock_t *lock);
/*Releases given lock and restores local interrupts to given previous state*/
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);

/*Tries to acquire given lock; if unavailable, returns nonzero*/
int spin_trylock(spinlock_t *lock);
/*Returns nonzero if the lock is currently acquired, otherwise returns zero*/
int spin_is_locked(spinlock_t *lock);
```

Kernel Synchronization

Semaphore: definition

Problems with spinlock

- busy waiting
- can't sleep when lock is acquired

信号量(semaphore): 整个critial section关闭中断太浪费，只在acquire lock时关闭中断

- Dijkstra 1965: Proberen (down, wait), Verhogen (up, signal)操作
- /include/linux/semaphore.h

```
struct semaphore {
    /* 保护count变量的自旋锁lock */
    raw_spinlock_t          lock;
    /* 信号量资源数量count */
    unsigned int          count;
    /* 等待队列wait_list */
    struct list_head     wait_list;
};
```

Semaphore: API

Kernel semaphore: 声明，初始化，down，up

```
DEFINE_SEMAPHORE(name);
/* or */
struct semaphore s = __SEMAPHORE_INITIALIZER(name, val);
struct semaphore s;
sem_init(&s, val);
down(&s);
/* also: down_{interruptible,killable,trylock,timeout} */
up(&s);
```

User semaphore: 声明，初始化，down，up

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0 /* process shared */, val);
sem_wait(&s);
sem_post(&s);
```

Semaphore: API

静态声明:

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

#define __SEMAPHORE_INITIALIZER(name, n)              \
{                                                            \
        .lock           = __RAW_SPIN_LOCK_UNLOCKED((name).lock),       \
        .count          = n,                                         \
        .wait_list      = LIST_HEAD_INIT((name).wait_list),          \
}
```

动态创建:

```
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);
}
```

Semaphore: API

<linux/semaphore.h>

```
/*acquire the semaphore, sleep if no semaphore available*/
void down(struct semaphore *sem);
/*release the semaphore*/
void up(struct semaphore *sem);

/*acquire the semaphore unless interrupted*/
int  down_interruptible(struct semaphore *sem);
/*acquire the semaphore unless killed*/
int  down_killable(struct semaphore *sem);
/*try to acquire the semaphore, without waiting*/
int  down_trylock(struct semaphore *sem);
/*acquire the semaphore within a specified time*/
int  down_timeout(struct semaphore *sem, long jiffies);
```

Semaphore: void down(struct semaphore *sem);

```
void down(struct semaphore *sem)
{
        unsigned long flags;

        raw_spin_lock_irqsave(&sem->lock, flags); /*保护sem->count*/
        if (likely(sem->count > 0))
                sem->count--;/*获得信号量*/
        else
                __down(sem);/*等待*/
        raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(down);

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

Semaphore: down

```c
/*uninterruptable sleep process will not be woken up by signals, can be
 *only woken up by what it's waiting for: wake_up_process().
 */
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);
}
```

```c
static inline int __sched __down_common(struct semaphore *sem, long state,
                                long timeout)
{
    struct semaphore_waiter waiter;

    list_add_tail(&waiter.list, &sem->wait_list);
    waiter.task = current;
    waiter.up = false;

    for (;;) {
        if (signal_pending_state(state, current))
            goto interrupted;
        if (unlikely(timeout <= 0))
            goto timed_out;
        __set_current_state(state);
        raw_spin_unlock_irq(&sem->lock);
        timeout = schedule_timeout(timeout); /*sleep until timeout*/
        raw_spin_lock_irq(&sem->lock);
        if (waiter.up)
            return 0;
    }

 timed_out:
    list_del(&waiter.list);
    return -ETIME;

 interrupted:
    list_del(&waiter.list);
    return -EINTR;
}
struct semaphore_waiter {
        struct list_head list;
        struct task_struct *task;
        bool up;
};
```

Semaphore: void up(struct semaphore *sem)

```
void up(struct semaphore *sem)
{
        unsigned long flags;

        raw_spin_lock_irqsave(&sem->lock, flags);
        if (likely(list_empty(&sem->wait_list)))
                sem->count++;
        else
                __up(sem);
        raw_spin_unlock_irqrestore(&sem->lock, flags);
}
EXPORT_SYMBOL(up);

static noinline void __sched __up(struct semaphore *sem)
{
        struct semaphore_waiter *waiter = list_first_entry(&sem->wait_list,
                                                struct semaphore_waiter, list);
        list_del(&waiter->list);
        waiter->up = true;
        wake_up_process(waiter->task);
}
```

Kernel Synchronization

- Synchronization
- 自旋锁: Spin lock
- 信号量: Semaphore
- 互斥锁: Mutex
- 读写锁: Read write lock
- RCU: Read-Copy-Update
- Lab3

Problems with semaphore

```
P(S);
//...Critical Section
V(S);
```

1. Accidental Release
2. Recursive Deadlock
3. Task-Death Deadlock
4. Priority Inversion

```
// Forgot... P(S);
//...Critical Section
V(S);
```

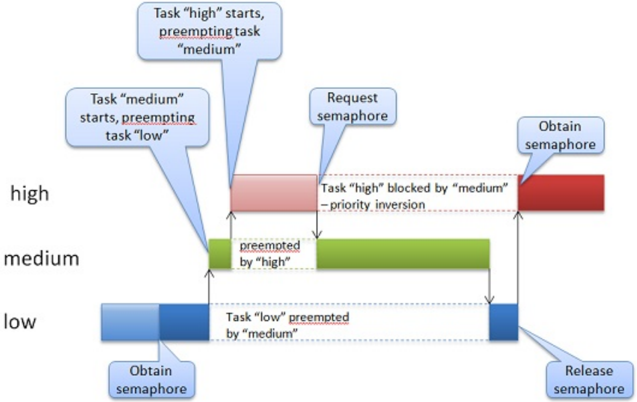
```
func A()
{
    P(S);
    A();
    V(S);
}
```

```
func A()          func B()
{                 {
 P(S);              P(S);
 //Death            //wait...
 //V(S);            //Deadlock
}                 }
```

Problems with semaphore

NASA Mars Pathfinder: the first bug on Mars.

The problem due to priority inversion caused the spacecraft to reset itself several times after it was landed on Mars on July 4, 1997.


Task "high" starts, preempting task "medium"
Task "medium" starts, preempting task "low"
Request semaphore
Obtain semaphore
high
Task "high" blocked by "medium" – priority inversion
medium
preempted by "high"
low
Task "low" preempted by "medium"
Obtain semaphore
Release semaphore

Mutex

Introduced by Dijkstra in 1980s. Mutex has more strict semantic.

- binary: only one process may hold mutex at one time
- only the owner of a mutex may release or unlock it

```
struct mutex {
        /* 1: unlocked, 0: locked, negative: locked, possible waiters */
        atomic_t                count;
        spinlock_t              wait_lock;
        struct list_head        wait_list;
#if defined(CONFIG_DEBUG_MUTEXES) || defined(CONFIG_MUTEX_SPIN_ON_OWNER)
        struct task_struct      *owner;
#endif
};

struct mutex_waiter {
        struct list_head        list;
        struct task_struct      *task;
#ifdef CONFIG_DEBUG_MUTEXES
        void                    *magic;
#endif
};
```

Mutex API

Kernel mutex: 声明，初始化，lock，unlock

```
DEFINE_MUTEX(name);
struct mutex s;
mutex_init(&s);
mutex_lock(&s);
/* also: mutex_lock_{interruptible,killable,io...}
mutex_unlock(&s);
```

Userspace: 声明，初始化，lock，unlock

```
#include <pthreads.h>
pthread_mutex_t m;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
/* or */
pthread_mutex_init(&m, attr);
pthread_mutex_lock(&m);
pthread_mutex_unlock(&m);
```

Linux mutex implementation

```
#define DEFINE_MUTEX(mutexname) \
        struct mutex mutexname = __MUTEX_INITIALIZER(mutexname)
#define __MUTEX_INITIALIZER(lockname)         \
{                                                                \
       .count = ATOMIC_INIT(1),                                  \
       .wait_lock = __SPIN_LOCK_UNLOCKED(lockname.wait_lock), \
       .wait_list = LIST_HEAD_INIT(lockname.wait_list)         \
}

# define mutex_init(mutex)                \
do {                                                       \
        static struct lock_class_key __key;             \
                                                           \
        __mutex_init((mutex), #mutex, &__key);         \
} while (0)

void __mutex_init(struct mutex *lock, const char *name, struct lock_class_key *key)
{
        atomic_set(&lock->count, 1);
        spin_lock_init(&lock->wait_lock);
        INIT_LIST_HEAD(&lock->wait_list);
        mutex_clear_owner(lock);
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
        osq_lock_init(&lock->osq);
#endif
        debug_mutex_init(lock, name, key);
}
```

Linux mutex implementation

```
void __sched mutex_lock(struct mutex *lock)
{
        might_sleep();
        __mutex_fastpath_lock(&lock->count, __mutex_lock_slowpath);
        mutex_set_owner(lock);
}
static inline void __mutex_fastpath_lock(atomic_t *v,
                                         void (*fail_fn)(atomic_t *))
{
        /*If the lock was FREE, the lock is acquired with only two instructions;
         *if the lock was BUSY, the code leaves count < 0 and invokes a separate
         *routine to handle the slow path.
         */
        asm_volatile_goto(LOCK_PREFIX "   decl %0\n"
                          "   jns %l[exit]\n" /*jump if not signed(if value is now 0)*/
                          : : "m" (v->counter)
                          : "memory", "cc"
                          : exit);
        fail_fn(v);
exit:
        return;
}
void __sched __mutex_lock_slowpath(atomic_t *lock_count)
{
        struct mutex *lock = container_of(lock_count, struct mutex, count);

        __mutex_lock_common(lock, TASK_UNINTERRUPTIBLE, 0,
                            NULL, _RET_IP_, NULL, 0);
}
```

Linux mutex implementation

```
static __always_inline int __sched
__mutex_lock_common(struct mutex *lock, long state, unsigned int subclass, /*...*/)
{
        struct task_struct *task = current;
        struct mutex_waiter waiter;
        preempt_disable();
        spin_lock_mutex(&lock->wait_lock, flags);
        list_add_tail(&waiter.list, &lock->wait_list);
        waiter.task = task;
        for (;;) {
                if (atomic_read(&lock->count) >= 0 &&
                    (atomic_xchg_acquire(&lock->count, -1) == 1))
                        break;
                /* this code get eliminated when uninterruptible*/
                if (unlikely(signal_pending_state(state, task))) {
                        ret = -EINTR;
                        goto err;
                }
                __set_task_state(task, state);
                /* didn't get the lock, go to sleep: */
                spin_unlock_mutex(&lock->wait_lock, flags);
                schedule_preempt_disabled();
                spin_lock_mutex(&lock->wait_lock, flags);
        }
        __set_task_state(task, TASK_RUNNING);
        mutex_remove_waiter(lock, &waiter, current_thread_info());
        /* set it to 0 if there are no waiters left: */
        if (likely(list_empty(&lock->wait_list)))
                atomic_set(&lock->count, 0);

err:
        spin_unlock_mutex(&lock->wait_lock, flags);
        preempt_enable();
        return ret;
}
```

Linux mutex implementation

```
void __sched mutex_unlock(struct mutex *lock)
{
    __mutex_fastpath_unlock(&lock->count, __mutex_unlock_slowpath);
}
static inline void __mutex_fastpath_unlock(atomic_t *v,
                                           void (*fail_fn)(atomic_t *))
{
       asm_volatile_goto(LOCK_PREFIX "   incl %0\n"
                         "   jg %l[exit]\n"   // jump if new value is 1
                         : : "m" (v->counter)
                         : "memory", "cc"
                         : exit);
       fail_fn(v);
exit:
       return;
}
void __mutex_unlock_slowpath(atomic_t *lock_count)
{
      struct mutex *lock = container_of(lock_count, struct mutex, count);
      __mutex_unlock_common_slowpath(lock, 1);
}
```

Linux mutex implementation

```
static inline void
__mutex_unlock_common_slowpath(struct mutex *lock, int nested)
{
    unsigned long flags;
    /*On the slow path, count was negative.*/
        if (__mutex_slowpath_needs_to_unlock())
                atomic_set(&lock->count, 1);
    spin_lock_mutex(&lock->wait_lock, flags);

    if (!list_empty(&lock->wait_list)) {
        /* get the first entry from the wait-list: */
        struct mutex_waiter *waiter =
                list_entry(lock->wait_list.next, struct mutex_waiter, list);

        wake_up_process(waiter->task);
    }

    spin_unlock_mutex(&lock->wait_lock, flags);
}
```

Mutex: How to fix priority inversion

设置pthread mutex的protocol属性为`PTHREAD_PRIO_INHERIT`。
Priority inheritance: increases the priority of a process (A) to the maximum priority of any other process waiting for any resource on which A has a resource lock (if it is higher than the original priority of A).

```
pthread_mutex_t m;
pthread_mutexattr_t ma;

pthread_mutexattr_init(&ma);
pthread_mutexattr_setprotocol(&ma, PTHREAD_PRIO_INHERIT);
pthread_mutex_init(&m, &ma);
```

Mutex API

<linux/mutex.h>

```
/*acquire the mutex*/
void mutex_lock(struct mutex *lock);
/*acquire the mutex, interruptible,  If a signal arrives while waiting for
 *the lock then this function returns -EINTR.*/
int mutex_lock_interruptible(struct mutex *lock);
/* acquire the mutex, interruptible only by signals that kill the process*/
int mutex_lock_killable(struct mutex *lock);

/*try to acquire the mutex, without waiting Returns 1 if the mutex has been
 *acquired successfully, and 0 on contention.*/
int mutex_trylock(struct mutex *lock);
void mutex_unlock(struct mutex *lock);
int mutex_is_locked(struct mutex *lock);
```

Kernel Synchronization

- Synchronization
- 自旋锁: Spin lock
- 信号量: Semaphore
- 互斥锁: Mutex
- 读写锁: Read write lock
- RCU: Read-Copy-Update
- Lab3

Read write lock

Some data is read way more often than written.

- some threads only read, others only write
- can allow multiple readers but only one writer
- $(nr \geq 0) \wedge (0 \leq nw \leq 1) \wedge ((nr \geq 0) \implies (nw = 0))$

How do we implement rwlock?

```
read_lock(L)
{
    lock(c);
    count++;
    if(count==1)
        lock(L);
    unlock(c);
}
```

```
read_unlock(L)
{
    lock(c);
    count--;
    if(count==0)
        unlock(L);
    unlock(c);
}
```

```
write_lock(L)
{
    lock(L);
}
```

```
write_unlock(L)
{
    unlock(L);
}
```

Read write lock in Linux

```
extern rwlock_t tasklist_lock;
```

```
DEFINE_RWLOCK(mr_rwlock);

read_lock(&mr_rwlock);
/* critical section (read only) ... */
read_unlock(&mr_rwlock);

write_lock(&mr_rwlock);
/* critical section (read and write) ... */
write_unlock(&mr_lock);
```

不能将读锁升级为写锁!

```
/* this will deadlock!*/
read_lock(&mr_rwlock);
/*...*/
write_lock(&mr_rwlock);
```

Read write spinlock API

<linux/rwlock.h>

```
/*Acquires given lock for reading;*/
void read_lock(rwlock_t *lock);
/*Disables local interrupts and acquires lock;*/
void read_lock_irq(rwlock_t *lock);
/*Saves the current state of local interrupts, then read_lock_irq()*/
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
/*Releases given lock for reading*/
void read_unlock(rwlock_t *lock);
/*Releases given lock and enables local interrupts;*/
void read_unlock_irq(rwlock_t *lock);
/* Releases given lock and restores local interrupts to the given state*/
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);

void write_lock(rwlock_t *lock);
void write_lock_irq(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_unlock(rwlock_t *lock);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);

/*Tries to acquire given lock for writing; if unavailable, returns nonzero;*/
int write_trylock(rwlock_t *lock);
```

Read write semaphore API

<linux/rwsem.h>

```
/*lock for reading*/
void down_read(struct rw_semaphore *sem);
/*try lock for reading;*/
int down_read_trylock(struct rw_semaphore *sem);
/*lock for writing;*/
void down_write(struct rw_semaphore *sem);
/*try lock for writing;*/
int down_write_trylock(struct rw_semaphore *sem);

/*release a read lock*/
void up_read(struct rw_semaphore *sem);
/* release a write lock*/
void up_write(struct rw_semaphore *sem);

static inline int rwsem_is_locked(struct rw_semaphore *sem);
void downgrade_write(struct rw_semaphore *sem);
```

Kernel Synchronization

Synchronization

自旋锁: Spin lock

信号量: Semaphore

互斥锁: Mutex

读写锁: Read write lock

RCU: Read-Copy-Update

Lab3

Read-Copy-Update (RCU)

RCU is a modern lock

- Busy waiting: 1962 Dijkstra
- Semaphore: 1965 Dijkstra, P(S) & V(S)
  - Dijkstra: `https://zh.forvo.com/word/dijkstra/#nl`
  - Carel S. Scholten: binary vs counting semaphore
- Mutex: 1980s, IEEE Std 1003.1/POSIX(1990)

Some data is read way more often than written

- Routing tables: consulted for each packet that is forwarded
- Read write lock/semaphore: reader lock contention - readers serialized in `read_lock()`, performance bottleneck.

```
read_lock(L)
{
    lock(c);
    count++;
    if(count==1)
        lock(L);
    unlock(c);
}
```

```
read_unlock(L)
{
    lock(c);
    count--;
    if(count==0)
        unlock(L);
    unlock(c);
}
```

```
write_lock(L)
{
    lock(L);
}
```

```
write_unlock(L)
{
    unlock(L);
}
```

Read-Copy-Update (RCU)

RCU: avoid reader lock contention.

- requirement: writer update needs to be atomic
- reader may only see the old value (before update)
- then only some reader see the new value (update happens)
- all reader see the new value (update is complete)

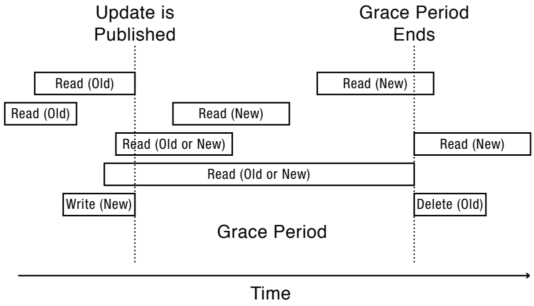


Figure 2: Timeline for an update concurrent with several reads for a data structure accessed with read-copy-update (RCU) synchronization.

Read-Copy-Update (RCU)

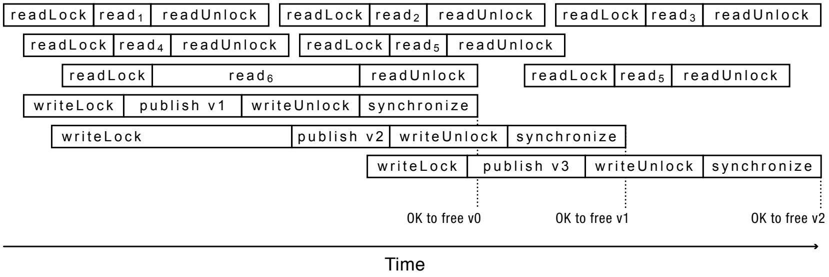


Figure 3: RCU allows one write at a time, and it allows reads to overlap each other and writes. The initial version is v0, and overlapping writes update the version to v1, v2, and then v3.

RCU lock example

```
typedef struct ElementS{
  int key;
  int value;
  struct ElementS *next;
} Element;

class RCUList {
  private:
    RCULock rcuLock;
    Element *head;
  public:
    bool search(int key, int *value);
    void insert(Element *item, value);
    bool remove(int key);
};
```

```
bool
RCUList::search(int key, int *valuep) {
    bool result = FALSE;
    Element *current;

    rcuLock.readLock();
    current = head;
    for (current = head; current != NULL;
                current = current->next) {
        if (current->key == key) {
            *valuep = current->value;
            result = TRUE;
            break;
        }
    }
    rcuLock.readUnlock();
    return result;
}
```

RCU lock example

```
void RCUList::insert(int key,
                     int value) {
    Element *item;

    // One write at a time.
    rcuLock.writeLock();

    // Initialize item.
    item = (Element*)
         malloc(sizeof(Element));
    item->key = key;
    item->value = value;
    item->next = head;

    //Atomically update list.
    //After this, reader may or
    //may not see the new item
    rcuLock.publish(&head, item);

    // Allow other writes
    // to proceed.
    rcuLock.writeUnlock();

    // Wait until no reader
    // has old version.
    rcuLock.synchronize();

    //any search will find new item
}
```

```
bool RCUList::remove(int key) {
    bool found = FALSE;
    Element *prev, *current;

    rcuLock.WriteLock(); //One write at a time.
    for (prev = NULL, current = head;
            current != NULL; prev = current,
            current = current->next) {
        if (current->key == key) {
            found = TRUE;

            // Publish update to readers
            if (prev == NULL) {
                rcuLock.publish(&head,
                          current->next);
            } else {
                rcuLock.publish(&(prev->next),
                          current->next);
            }
            break;
        }
    }
    rcuLock.writeUnlock(); //Allow other writes

    // Wait until no reader has old version.
    if (found) {
        rcuLock.synchronize();
        free(current);
    }
    return found;
}
```

RCU lock implementation [Paul McKenney]

```
class RCULock{
  private:
    // Global state
    Spinlock globalSpin;
    long globalCounter;
    // One per processor
    static long quiescentCount[NCPUs];
    // Per-lock state
    Spinlock writerSpin;
}

void RCULock::ReadLock() {
    disableInterrupts();
}

void RCULock::ReadUnlock() {
    enableInterrupts();
}

// called regularly by kernel scheduler
void RCULock::QuiescentState() {
    memory_barrier();
    for (int p=0; p<NCPUs; p++)
        quiescentCount[p] = globalCounter;
    memory_barrier();
}
```

```
void RCULock::writeLock() {
    writerSpin.acquire();
}

void RCULock::writeUnlock() {
    writerSpin.release();
}

void RCULock::publish (void **pp1, void *p2){
    memory_barrier();
    *pp1 = p2;
    memory_barrier();
}

//wait until there're no readers who
//started before you wrote
void RCULock::synchronize() {
    int p,c;
    globalSpin.acquire();
    c = ++globalCounter;
    globalSpin.release();
    for (p=0; p<NCPUs; p++)
    {
        while (quiescentCount[p] < c)
            sleep(10);
    }
}
```

Linux RCU API

RCU can only be used if you can publish your update as one change: `publish(p,v)`/`rcu_assign_pointer(p,v)`.
<linux/rculist.h>

```
Reader API
static inline void rcu_read_lock(void);
static inline void rcu_read_unlock(void);
rcu_dereference(p);
list_for_each_entry_rcu();

Writer API
/*Atomically update shared data structure*/
rcu_assign_pointer(p,v);
/*Wait for all currently active readers to exit critical section*/
static inline void synchronize_rcu(void);
list_add_rcu();
list_add_tail_rcu()
list_del_rcu();
```

No write lock implemented because any lock would be OK to use to synchronize writer.

Linux RCU usage

```c
struct task_struct *copy_process(/*..*/)
{
        struct task_struct *p;
        /* Make visible to the rest of the system. Need tasklist lock for parent etc handling!*/
        write_lock_irq(&tasklist_lock);
        if (likely(p->pid)) {
                init_task_pid(p, PIDTYPE_PID, pid);
                if (thread_group_leader(p)) {
                        init_task_pid(p, PIDTYPE_PGID, task_pgrp(current));
                        init_task_pid(p, PIDTYPE_SID, task_session(current));

                        list_add_tail(&p->sibling, &p->real_parent->children);
                        list_add_tail_rcu(&p->tasks, &init_task.tasks);
                        __this_cpu_inc(process_counts);
                } else {
                        current->signal->nr_threads++;
                        atomic_inc(&current->signal->live);
                        atomic_inc(&current->signal->sigcnt);
                        list_add_tail_rcu(&p->thread_group,
                                          &p->group_leader->thread_group);
                        list_add_tail_rcu(&p->thread_node,
                                          &p->signal->thread_head);
                }
                nr_threads++;
        }
        total_forks++;
        write_unlock_irq(&tasklist_lock);
        return p;
}
```

总结

Kernel synchronization

- Atomic operations: reference counter
- Spinlock: locks acquired for a short time
- Semaphore: avoid busy waiting, can be used for signalling
- Mutex: binary "semaphore" with owner
- RCU: modern locks optimized for readers (lock free)

Others

- Per-CPU variables: avoid the need for synchronizations
- functional programming languages
- memory barrier
- ...

Kernel Synchronization

- Synchronization
- 自旋锁: Spin lock
- 信号量: Semaphore
- 互斥锁: Mutex
- 读写锁: Read write lock
- RCU: Read-Copy-Update
- Lab3

Lab3: Linux kernel synchronization mechanism

Build a new kernel synchronization mechanism (a new lock) based on device orientation.
Implement the following four system calls.

```
/*
 * Sets current device orientation in the kernel.
 * System call number 326.
 */
int set_orientation(struct dev_orientation *orient);

struct dev_orientation {
    int azimuth; /* angle between the magnetic north and the Y axis,
              *  around the Z axis (-180<=azimuth<=180)*/
    int pitch;   /* rotation around the X-axis: -90<=pitch<=90 */
    int roll;    /* rotation around Y-axis: +Y == -roll, -180<=roll<=180 */
};
```

Lab3: Linux kernel synchronization mechanism

```c
/*
 * Create a new orientation event using the specified orientation range.
 * Return an event_id on success and appropriate error on failure.
 * System call number 327.
 */
int orientevt_create(struct orientation_range *orient);

/*
 * Destroy an orientation event and notify any processes which are
 * currently blocked on the event to leave the event.
 * Return 0 on success and appropriate error on failure.
 * System call number 328.
 */
int orientevt_destroy(int event_id);

/*
 * Block a process until the given event_id is notified. Verify that the
 * event_id is valid.
 * Return 0 on success and appropriate error on failure.
 * System call number 329.
 */
int orientevt_wait(int event_id);
```

Lab3: Linux kernel synchronization mechanism

写两个C程序来测试系统。两个程序分别fork n个child（n>2）。每个child都等待同一个事件，并执行以下操作：

- faceup：当模拟器屏幕朝向你时每隔一秒打印`"%d: facing up!"`，其中`"%d"`是该进程在child集合中的序号。
- facedown：当模拟器屏幕背向你时每隔一秒打印`"%d: facing down!"`，其中`"%d"`是该进程在child集合中的序号。

同时父进程等待60秒，然后通过关闭打开的方向事件来关闭所有子进程（而不是通过发送信号或其他此类方法）。

Lab3: Linux kernel synchronization mechanism

Sleeping via wait queues: A wait queue is a list of processes waiting for an event to occur, wait queues are represented in the kernel by `wake_queue_head_t`.

- created statically via `DECLARE_WAITQUEUE()`
- created dynamically via `init_waitqueue_head()`

Processes put themselves on a wait queue and mark themselves not runnable. When the event associated with the wait queue occurs, the processes on the queue are awakened.

```
/* 'q' is the wait queue we wish to sleep on */
DEFINE_WAIT(wait);

add_wait_queue(q, &wait);
while (!condition) { /* condition is the event that we are waiting for */
    prepare_to_wait(&q, &wait, TASK_INTERRUPTIBLE);
    if (signal_pending(current))
        /* handle signal */
    schedule();
}
finish_wait(&q, &wait);
```

Lab3: Linux kernel synchronization mechanism

Sleeping via wait queues

1. Creates a wait queue entry via the macro `DEFINE_WAIT()`.
2. Adds itself to a wait queue via `add_wait_queue()`. This wait queue awakens the process when the condition for which it is waiting occurs. Of course, there needs to be code elsewhere that calls `wake_up()` on the queue when the event actually does occur.
3. Calls `prepare_to_wait()` to change the process state to either `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`. This function also adds the task back to the wait queue if necessary, which is needed on subsequent iterations of the loop.
4. If the state is set to `TASK_INTERRUPTIBLE`, a signal wakes the process up. This is called a spurious wake up (a wake-up not caused by the occurrence of the event). So check and handle signals.
5. When the task awakens, it again checks whether the condition is true. If it is, it exits the loop. Otherwise, it again calls `schedule()` and repeats.

Thanks!

基础实验楼乙124