Linux Kernel Synchronization Primitives

April 4, 2018

Byung-Gon Chun

Acknowlegments. Some slides and/or picture in the following are adapted from UW, Columbia, and UC Berkeley class slides

Linux kernel synchronization primitives

- Memory barriers
  – avoids compiler, CPU instruction re-ordering
- Atomic operations
  – Read-modify-write ops
- RCU
  – Atomic pointer update, list APIs
- Interrupt/softirq disabling/enabling
- Spin locks
  – general, read/write
- Semaphores/Mutex
  – general, read/write, mutex
- Completion
- Waitqueue
- Seq Locks
  – provides reader side transactional memory

Choosing synch primitives

- Avoid synch if possible! (clever instruction ordering)
  - Example: RCUs
- Use atomics or rw spinlocks if possible
- Use semaphores or mutexes if you need to sleep
  - Can’t sleep in interrupt context
  - Don’t sleep holding a spinlock!
- Complicated matrix of choices for protecting data structures accessed by deferred functions

Atomic operations

- Many instructions not atomic in hardware
  - Read-modify-write instructions: inc, test-and-set, swap
  - Unaligned memory access
- Compiler may not generate atomic code
  - even i++ is not necessarily atomic!
- If the data that must be protected is a single word, atomic operations can be used. These functions examine and modify the word atomically.

Linux kernel atomic operations

- The atomic integer data type is *atomic_t*.
  - The atomic operations are used only with these special types

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

* atomic64_t

- A common use of the atomic integer operations:
  - Counters
  - Atomically performing an operation and testing
- Atomicity vs. ordering

Atomic integer operations

ATOMIC_INIT – initialize an *atomic_t* variable (integer)

atomic_read – examine value atomically

atomic_set – change value atomically

atomic_inc – increment value atomically

atomic_dec – decrement value atomically

atomic_add - add to value atomically

atomic_sub – subtract from value atomically

atomic_inc_and_test – increment value and test for zero

atomic_dec_and_test – decrement value and test for zero

atomic_sub_and_test – subtract from value and test for zero

* 64 bit integer and bitwise operations are also available

Atomic bitwise operations

```
unsigned long word = 0;

set_bit(0, &word);
set_bit(1, &word);
clear_bit(1, &word);
change_bit(0, &word);
if (test_and_set_bit(0, &word)) {
}

find_first_bit(unsigned long *addr, unsigned int size);
find_first_zero_bit(unsigned long *addr, unsigned int size);
```

Barrier operations

- *barrier* – prevent only compiler reordering
- *mb* – prevents load and store reordering
- *rmb* – prevents load reordering
- *wmb* – prevents store reordering
- *smp_mb* – prevent load and store reordering only in SMP kernel
- *smp_rmb* – prevent load reordering only in SMP kernels
- *smp_wmb* – prevent store reordering only in SMP kernels
- *set_mb* – performs assignment and prevents load and store reordering

Interrupt operations

- Intel: “interrupts enabled bit”
  - cli to clear (disable), sti to set (enable)

- Services used to serialize with interrupts are:

  local_irq_disable - disables interrupts on the current CPU

  local_irq_enable - enable interrupts on the current CPU

  local_save_flags - return the interrupt state of the processor

  local_restore_flags - restore the interrupt state of the processor

- Dealing with the full interrupt state of the system is officially discouraged. Locks should be used.

Spin locks

- A spin lock is a data structure (*spinlock_t*) that is used to synchronize access to critical sections.

- Only one thread can be holding a spin lock at any moment. All other threads trying to get the lock will "spin" (loop while checking the lock status).

- Spin locks should not be held for long periods because waiting tasks on other CPUs are spinning, and thus wasting CPU execution time.

CSE 컴퓨터공학부 Department of Computer Science & Engineering

Spin lock operations

- Functions used to work with spin locks (struct spinlock_t):

DEFINE_SPINLOCK – initialize a spin lock before using it for the first time

spin_lock – acquire a spin lock, spin waiting if it is not available

spin_unlock – release a spin lock

spin_unlock_wait – spin waiting for spin lock to become available, but don't acquire it

spin_trylock – acquire a spin lock if it is currently free, otherwise return error

spin_is_locked – return spin lock state

Spin locks & interrupts

- The spin lock services also provide interfaces that serialize with interrupts (on the current processor):

spin_lock_irq - acquire spin lock and disable interrupts

spin_unlock_irq - release spin lock and reenable

spin_lock_irqsave - acquire spin lock, save interrupt state, and disable

spin_unlock_irqrestore - release spin lock and restore interrupt state

RW spin lock operations

- Several functions are used to work with read/write spin locks (struct rwlock_t):

DEFINE_RWLOCK, rwlock_init – initialize a read/write lock before using it for the first time

read_lock – get a read/write lock for read

write_lock – get a read/write lock for write

read_unlock – release a read/write lock that was held for read

write_unlock – release a read/write lock that was held for write

read_trylock, write_trylock – acquire a read/write lock if it is currently free, otherwise return error

RW spin locks & interrupts

- The read/write lock services also provide interfaces that serialize with int errupts
  (on the current processor):

read_lock_irq - acquire lock for read and disable interrupts

read_unlock_irq - release read lock and reenable

read_lock_irqsave - acquire lock for read, save interrupt state, and disable

read_unlock_irqrestore - release read lock and restore interrupt state

- Corresponding functions for write exist as well (e.g., *write_lock_irqsave*).

CSE 컴퓨터공학부
Department of Computer Science & Engineering

Wait queue

- While writing modules there might be situations where one might have to wait for input some condition to occur before proceeding further.
- To manage a wait queue ,we need a structure of the kind wait_queue_head_t, which is defined in linux/wait.h.
- Once the wait queue has been created, we can put a task to sleep on the queue we created using one of the following.
  wait_event("queue","condition")
  wait_event_interruptible("queue","condition")
  wait_event_timeout("queue","condition","timeout")
  wait_event_interruptible_timeout("queue","condition","timeout")
- Once a task has been put to sleep we need to wake it up , which can be done using following :
  wake_up(queue)
  wake_up_interruptible (queue)

Semaphores

- A *semaphore* is a data structure that is used to synchronize access to critical sections or other resources.

- A *semaphore* allows a fixed number of tasks (generally one for critical sections) to "hold" the semaphore at one time. Any more tasks requesting to hold the *semaphore* are blocked (put to sleep).

Semaphore operations

- Operations for manipulating semaphores:

up – release the semaphore

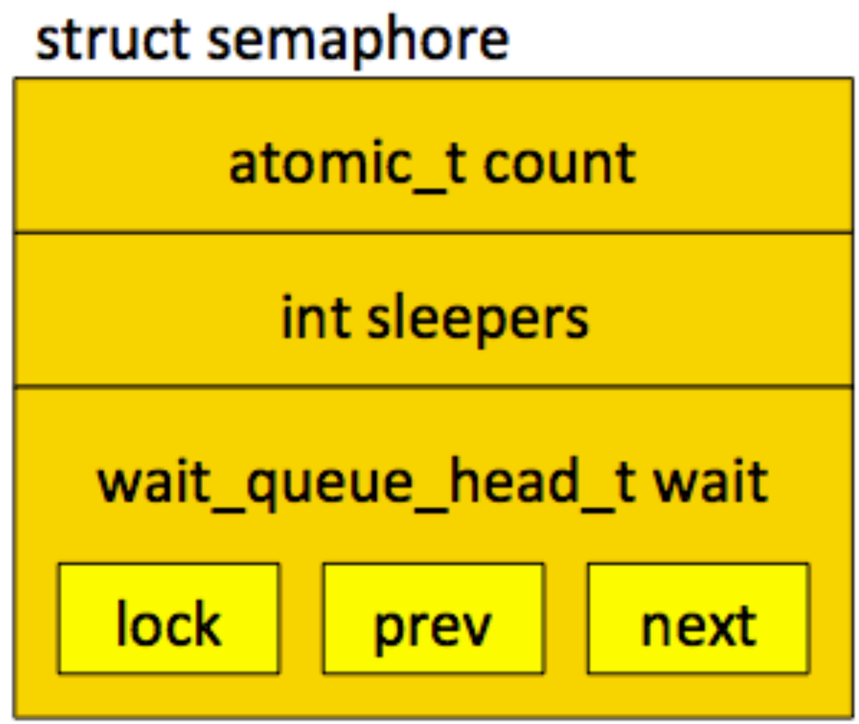
down – get the semaphore (can block)

down_interruptible – get the semaphore, but the operation is interruptible

down_trylock – try to get the semaphore without blocking, otherwise return an error

Semaphore structure

- struct semaphore
  - count (atomic_t):
    - > 0: free;
    - = 0: in use, no waiters;
    - < 0: in use, waiters
  - wait: wait queue
  - sleepers:
    - 0 (none),
    - 1 (some), occasionally 2
- Implementation requires lower-level synch
  - atomic updates, spinlock, interrupt disabling



RW semaphores

- A *rw_semaphore* is a semaphore that allows either one writer or any number of readers (but not both at the same time) to hold it.
- Any writer requesting to hold the *rw_semaphore* is blocked when there are readers holding it.
- A *rw_semaphore* can be used for serialization only in code that is allowed to block. Both types of semaphores are the only synchronization objects that should be held when blocking.
- Writers will not starve: once a writer arrives, readers queue behind it
- Increases concurrency

RW Semaphore Operations

- Operations for manipulating semaphores:

up_read – release a *rw_semaphore* held for read.

up_write – release a *rw_semaphore* held for write.

down_read – get a *rw_semaphore* for read (can block, if a writer is holding i t)

down_write – get a *rw_semaphore* for write (can block, if one or more read ers are holding it)

More RW Semaphore Ops

- Operations for manipulating semaphores:

down_read_trylock – try to get a *rw_semaphore* for read without blocking, otherwise return an error

down_write_trylock – try to get a *rw_semaphore* for write without blocking, otherwise return an error

downgrade_write – atomically release a *rw_semaphore* for write and acquire it for read (can't block)

CSE 컴퓨터공학부
Department of Computer Science & Engineering

Mutexes

- A *mutex* is a data structure that is *also* used to synchronize access to critical sections or other resources, introduced in 2.6.16.
- Why? (Documentation/mutex-design.txt)
  - simpler (lighter weight)
  - lighter code
  - slightly faster, better scalability
  - no fastpath tradeoffs?
  - debug support – strict checking of adhering to semantics
- Prefer mutexes over semaphores

Mutex Operations

- Operations for manipulating mutexes:

mutex_unlock – release the mutex

mutex_lock – get the mutex (can block)

mutex_lock_interruptible – get the mutex, but allow interrupts

mutex_trylock – try to get the mutex without blocking, otherwise return an error

mutex_is_locked – determine if mutex is locked

Completions

- Slightly higher-level, FIFO semaphores

- Up/down may execute concurrently
  - This is a good thing (when possible)

- Operations: complete(), wait_for_complete()
  - Spinlock and wait_queue
  - Spinlock serializes ops
  - Wait_queue enforces FIFO

Linux Kernel Seq Locks

- Locks that favor writers over readers
  - Lots of readers, few writers, light-weight
  - Programmer invoked transactional memory
  - Limited – doesn't support lock free concurrent writes
- Basic idea:
  - Lock is associated with sequence number
  - Writers increment seq number
  - Readers check seq number at lock and unlock
  - If different, try again
  - Writers synchronize between themselves, never block for readers

Seq Lock Operations

- Operations for manipulating seq locks:

DEFINE_SEQLOCK – initialize seq lock

write_seqlock – get the seqlock as writer, incr seq (can block)

write_sequnlock – release seqlock, incr seq

read_seqbegin, *read_seqretry* – define read atomic region, seqretry returns true if op was atomic

Writer

```
write_seqlock(&mr_seq_lock);
/* update data here */
write_sequnlock(&mr_seq_lock);
```

Reader

```
do {
    seq = read_seqbegin (&mr_seq_lock);
    /* read data here */
} while (read_seqretry(&mr_seq_lock, seq));
```