W4118 Operating Systems

Instructor: Junfeng Yang
Learning goals of this lecture

- Different flavors of synchronization primitives and when to use them, in the context of Linux kernel
- How synchronization primitives are implemented for real
- “Portable” tricks: useful in other context as well (when you write a high performance server)
  - Optimize for common case
Synchronization is complex and subtle

- Already learned this from the code examples we’ve seen

- Kernel synchronization is even more complex and subtle
  - Higher requirements: performance, protection …
  - Code heavily optimized, “fast path” often in assembly, fit within one cache line
Recall: Layered approach to synchronization

- Hardware provides simple low-level atomic operations, upon which we can build high-level, synchronization primitives, upon which we can implement critical sections and build correct multi-threaded/multi-process programs.
Outline

- Low-level synchronization primitives in Linux
  - Memory barrier
  - Atomic operations
  - Synchronize with interrupts
  - Spin locks

- High-level synchronization primitives in Linux
  - Completion
  - Semaphore
  - Futex
  - Mutex
Architectural dependency

- Implementation of synchronization primitives: highly architecture dependent

- Hardware provides atomic operations

- Most hardware platforms provide test-and-set or similar: examine and modify a memory location atomically

- Some don’t, but would inform if operation attempted was atomic
Memory barrier motivation

- Evil compiler!
  - Reorder code as long as it correctly maintains data flow dependencies within a function and with called functions

- Evil hardware!
  - Reorder instruction execution as long as it correctly maintains register flow dependencies
  - Reorder memory modification as long as it correctly maintains data flow dependencies
Memory barrier definition

- **Memory Barriers**: instructions to compiler and/or hardware to complete all pending accesses before issuing any more
  - Prevent compiler/hardware reordering

- **Read memory barriers**: prevent reordering of read instructions

- **Write memory barriers**: prevent reordering of write instructions
Linux 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

- include/asm-i386/system.h
Outline

- Low-level synchronization primitives in Linux
  - Memory barrier
  - Atomic operations
  - Synchronize with interrupts
  - Spin locks

- High-level synchronization primitives in Linux
  - Completion
  - Semaphore
  - Mutex
  - Futex
Atomic operations

- Some instructions not atomic in hardware (smp)
  - Read-modify-write instructions that touch memory twice, e.g., inc, xchg

- Most hardware provides a way to make these instructions atomic
  - Intel lock prefix: appears to lock the memory bus
  - Execute at memory speed
Linux atomic operations

- **ATOMIC_INIT** - initialize an `atomic_t` variable
- **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 from value and test for zero
- **atomic_sub_and_test** - subtract from value and test for zero
- **atomic_set_mask** - mask bits atomically
- **atomic_clear_mask** - clear bits atomically

- `include/asm-i386/atomic.h`
Outline

- Low-level synchronization primitives in Linux
  - Memory barrier
  - Atomic operations
  - Synchronize with interrupts
  - Spin locks

- High-level synchronization primitives in Linux
  - Completion
  - Semaphore
  - Futex
  - Mutex
Linux interrupt operations

- `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.
Outline

- Low-level synchronization primitives in Linux
  - Memory barrier
  - Atomic operations
  - Synchronize with interrupts
  - Spin locks

- High-level synchronization primitives in Linux
  - Completion
  - Semaphore
  - Mutex
  - Futex
Linux spin lock operations

- **spin_lock_init** - 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

- `include/asm-i386/spinlock.h` and `kernel/spinlock.c`
Spin lock usage rules

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

- Remember, don’t call blocking operations (any function that may call `schedule()`) when holding a spin lock.
Linux spin lock implementation

__raw_spin_lock_string

1: lock; decb %0  # atomically decrement
   jns 3f       # if clear sign bit (>=0) jump forward to 3
2: rep; nop    # wait
   cmpb $0, %0  # spin - compare to 0
   jle 2b       # go back to 2 if <= 0 (locked)
   jmp 1b       # unlocked; go back to 1 to try again
3:            # we have acquired the lock …

spin_unlock merely writes 1 into the lock field.
Variant of spin locks and operations

- Spin locks that serialize with interrupts
- Read-write spin locks (*rwlock_t*)
- Read-write spin locks that serialize with interrupts
- Big reader lock (*brlock*)
- Sequential lock (*seqlock*)
Outline

- **Low-level synchronization primitives in Linux**
  - Memory barrier
  - Atomic operations
  - Synchronize with interrupts
  - Spin locks

- **High-level synchronization primitives in Linux**
  - Completion
  - Semaphore
  - Futex
  - Mutex
Completions

- Simple way to ensure execution order: wait and wake-up semantics

- `wait_for_complete(struct completion*)` - wait for another thread to call `complete()`

- `compute(struct completion*)` - wake up threads waiting inside `wait_for_complete()`

- Implemented using spinlock and `wait_queue`

- `include/linux/completion.h` and `kernel/sched.c`
Outline

- Low-level synchronization primitives in Linux
  - Memory barrier
  - Atomic operations
  - Synchronize with interrupts
  - Spin locks

- High-level synchronization primitives in Linux
  - Completion
  - Semaphore
  - Futex
  - Mutex
Linux semaphore operations

- **up** - release the semaphore
- **down** - get the semaphore (can block)
- **down_interruptible** - get the semaphore, can be woken up if interrupt arrives
- **down_trylock** - try to get the semaphore without blocking, otherwise return an error

- include/asm-i386/semaphore.h and arch/i386/kernel/semaphore.c
Linux semaphore implementation

- **Goal:** Optimize for uncontended (common) case

- **Implementation idea**
  - Uncontended case: use atomic operations
  - Contended case: use spin locks and wait queues
Linux semaphore structure

- **Struct semaphore**
  - **count (atomic_t):**
    - > 0: free;
    - = 0: in use, no waiters;
    - < 0: in use, waiters
  - **sleepers:**
    - 0 (none)
    - 1 (some), occasionally 2
  - **wait: wait queue**

- implementation requires lower-level synchronization primitives
  - atomic updates, spinlock, interrupt disabling
Contrived (buggy) semaphore implementation

up (struct semaphore* s) {
    if(atomic_inc_positive(&s->count))
        return;
    wake_up(&s->wait);
}

down (struct semaphore *s) {
    if(!atomic_dec_negative(&s->count))
        return; // uncontended
    // contended
    add_wait_queue_exclusive(&s->wait, self);
}

- Common case: only one atomic instruction
- Problem
  - Concurrent calls to up() and down()?
  - Concurrent calls to down()?
The real down()

```c
inline down:
    movl $sem, %ecx # why does this work?
    lock; decl (%ecx)# atomically decr sem count
    jns 1f          # if not negative jump to 1
    lea %ecx, %eax # move into eax
    call __down_failed #
1:                # we have the semaphore
```

```c
#include/asm-i386/semaphore.h and arch/i386/kernel/semaphore.c
```

```c
down_failed:
    pushl %edx       # push edx onto stack (C)
    pushl %ecx       # push ecx onto stack
    call __down      # call C function
    popl %ecx        # pop ecx
    popl %edx        # pop edx
    ret
```
__down()

tsk->state = TASK_UNINTERRUPTIBLE;
spin_lock_irqsave(&sem->wait.lock, flags);
add_wait_queue_exclusive_locked(&sem->wait, &wait);
sem->sleepers++;
for (;;) {
    int sleepers = sem->sleepers;
    /*
       * Add "everybody else" into it. They aren't playing,
       * because we own the spinlock in the wait_queue head
       */
    if (!atomic_add_negative(sleepers - 1, &sem->count)) {
        sem->sleepers = 0;
        break;
    }
    sem->sleepers = 1;  /* us - see -1 above */
    spin_unlock_irqrestore(&sem->wait.lock, flags);
    schedule();
    spin_lock_irqsave(&sem->wait.lock, flags);
    tsk->state = TASK_UNINTERRUPTIBLE;
}
remove_wait_queue_locked(&sem->wait, &wait);
wake_up_locked(&sem->wait);
spin_unlock_irqrestore(&sem->wait.lock, flags);
tsk->state = TASK_RUNNING;

Linux/lib/semaphore-sleepers.c
Outline

- Low-level synchronization primitives in Linux
  - Memory barrier
  - Atomic operations
  - Synchronize with interrupts
  - Spin locks

- High-level synchronization primitives in Linux
  - Completion
  - Semaphore
  - Futex
  - Mutex
Futex motivation

- Synchronization of kernel-level threads: expensive
  - Each synchronization operation traps into kernel

- Futex: optimize for the uncontended (common) case
  - Uncontended $\rightarrow$ no kernel involvement
  - Contended $\rightarrow$ trap into kernel
Futex implementation

- Borrows from Linux kernel semaphore implementation

- Data structure
  - An aligned integer in user space
  - A wait queue in kernel space

- Operations
  - Uncontended case: atomic operations, user space
  - Contended case: spin locks and wait queues, kernel space

- pthread mutex, semaphore, and condition variables use futex
Outline

- Low-level synchronization primitives in Linux
  - Memory barrier
  - Atomic operations
  - Synchronize with interrupts
  - Spin locks

- High-level synchronization primitives in Linux
  - Completion
  - Semaphore
  - Futex
  - Mutex
Linux mutexes: motivation

- Introduced in 2.6.16.

- “90% of semaphores in kernel are used as mutexes, 9% of semaphores should be spin_locks.” Andrew Morton

- Slow paths are more critical for highly contended semaphores on SMP

- Mutexes are simpler
Linux mutex operations

- `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