Kernel Critical Sections

Real Time Operating Systems and Middleware

Luca Abeni

luca.abeni@unitn.it

Critical Sections in Kernel Code

- Old Linux kernels used to be non-preemptable...
- Kernel $\Rightarrow$ Big critical section
- Mutual exclusion was not a problem...
- Then, multiprocessor systems changed everything
  - First solution: Big Kernel Lock $\leftarrow$ very bad!
- Removed BKL, and preemptable kernels, ...
  - Multiple tasks can execute inside the kernel simultaneously $\Rightarrow$ mutual exclusion is an issue!
  - Multiple critical sections inside the kernel

Enforcing Mutual Exclusion

- Mutual exclusion is traditionally enforced using mutexes
- Mutexes are blocking synchronisation objects
  - A task trying to acquire a locked mutex is blocked. . .
  - . . .And the scheduler is invoked!
- Good solution for user-space applications...
- But blocking is sometimes bad when in the kernel!

Blocking is Bad When...

- **Atomic Context**
  - Code in “task” context can sleep (task blocked)
  - . . .But some code does not run in a task context (example: IRQ handlers)!
  - Other situations (ex: interrupts disabled)
- **Efficiency**
  - small critical sections $\rightarrow$ using mutexes, a task would block for a very short time
  - Busy-waiting can be more efficient (less context switches)!

Summing up...

- In some particular situations. . .
- . . .We need a way to enforce mutual exclusion *without blocking* any task
  - This is only useful in kernel programming
  - Remember: in general cases, busy-waiting is bad!
- So, the kernel provides a *spinning lock* mechanism
  - To be used when sleeping/blocking is not an option
  - Originally developed for multiprocessor systems

Spinlocks - The Origin

- spinlock: Spinning Lock
  - Protects shared data structures in the kernel
  - Behaviour: similar to mutex (*locked* / *unlocked*)
  - But does not sleep!
- Basic idea: busy waiting (spin instead of blocking)
- Might neeed to disable interrupts in some cases

Spinlocks - Operations

- Basic operations on spinlocks: similar to mutexes
    - Biggest difference: `lock()` on a locked spinlock
- `lock()` on an unlocked spinlock: change its state
- `lock()` on a locked spinlock: spin until it is unlocked
    - Only useful on multiprocessor systems
- `unlock()` on a locked spinlock: change its state
- `unlock()` on an unlocked spinlock: error!!!

Spinlocks - Implementation

```
int lock = 1;

void lock(int *sl)
{
  while (TestAndSet(sl, 0) == 0);
}

void unlock(int *sl)
{
  *sl = 1;
}
```

A possible algorithm (using test and set)

```
    lock:
      decb %0
      jns 3
    2:
      cmpb $0,%0
      jle 2
      jmp lock
    3:
      ...
    unlock:
      movb $1,%0
```

Assembler implementation (in Linux)

Spinlocks and Livelocks

- Trying to lock a locked spinlock results in spinning
  $\Rightarrow$ spinlocks must be locked for a **very short** time
- If an interrupt handler interrupts a task holding a spinlock, livelocks are possible...
  - τ_i gets a spinlock SL
  - An interrupt handler interrupts τ_i...
  - ...And tries to get the spinlock SL
  - $\Rightarrow$ The interrupt handler spins waiting for SL
  - But τ_i cannot release it!!!

Avoiding Livelocks

- Resource shared with ISRs $\rightarrow$ possible livelocks
  - What to do?
  - The ISR should not run during the critical section!
- When a spinlock is used to protect data structures shared with interrupt handlers, the spinlock must disable the execution of such handlers!
  - In this way, the kernel cannot be interrupted when it holds the spinlock!

Spinlocks in Linux

- Defining a spinlock: `spinlock_t my_lock;`
- Initialising: `spin_lock_init(&my_lock);`
- Acquiring a spinlock: `spin_lock(&my_lock);`
- Releasing a spinlock: `spin_unlock(&my_lock);`
- With interrupt disabling:
  - `spin_lock_irq(&my_lock)`, `spin_lock_bh(&my_lock)`, `spin_lock_irqsave(&my_lock, flags)`
  - `spin_unlock_irq(&my_lock)`, ...

Spinlocks - Evolution

- On UP systems, traditional spinlocks are no-ops
  - The `_irq` variations are translated in `cli`/`sti`
- This works assuming only on execution flow in the kernel $\Rightarrow$ non-preemptable kernel
- Kernel preemptability changes things a little bit:
  - Preemption counter, initialised to 0: number of spinlocks currently locked
  - `spin_lock()` increases the counter
  - `spin_unlock()` decreases the counter

Spinlocks and Kernel Preemption

- preemption counter: increased when entering a critical section, decreased on exit
- When exiting a critical section, check if the scheduler can be invoked
  - If the preemption counter returns to 0, `spin_unlock()` calls `schedule()`...
  - ...And returns to user-space!
- Preemption can only happen on `spin_unlock()` (interrupt handlers lock/unlock at least one spinlock...)

Spinlocks and Kernel Preemption

- In preemptable kernels, spinlocks' behaviour changes a little bit:
  - `spin_lock()` disables preemption
  - `spin_unlock()` might re-enable preemption (if no other spinlock is locked)
  - `spin_unlock()` is a preemption point
- Spinlocks are not optimised away on UP anymore
- Become similar to mutexes with the Non-Preemptive Protocol (NPP)
- Again, they must be held for very short times!!!

Sleeping in Atomic Context

- *atomic context*: CPU context in which it is not possible to modify the state of the current task
  - Interrupt handlers
  - Scheduler code
  - Critical sections protected by spinlocks
  - . . .
- Do not call possibly-blocking functions from atomic context!!!