Sistemi Operativi 2

Kernel Locking

Luca Abeni

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, SMPs and preemptable kernels changed everything
  - Multiple tasks can execute inside the kernel simultaneously $\Rightarrow$ mutual exclusion is an issue!
  - Mutual exclusion can be enforced through mutexes
- Mutexes are blocking synchronisation objects
  - A task trying to acquire a locked mutex is blocked...
  - ...And the scheduler is invoked!
  - Blocking is sometimes bad

Blocking is Bad when...

- **Atomic Context**
  - Code running in a proper “task” context can sleep (so, the task can be blocked). . .
  - . . .But sometimes the code is not executing in a task context (example: IRQ handlers)!
  - In some other situations, a task cannot sleep even if it has a proper context (example: interrupt disabled)
- **Efficiency**
  - Sometimes, critical sections are very small $\rightarrow$ using mutexes, a task would block for a very short time
  - Busy-waiting can be more efficient, because it reduces the number of 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
  - Used to protect shared data structure in the kernel
  - Behaviour: similar to mutex (*locked* / *unlocked*)
  - But does not sleep!
- `lock()` on an unlocked spinlock: change its state
- `lock()` on a locked spinlock: spin until the mutex is unlocked
  - Only useful on multiprocessor systems
- `unlock()` on a locked spinlock: change its state
- `unlock()` on an unlocked spinlock: error!!!

Spinlocks - Implementation

```c
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 - Constraints

- 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, deadlocks 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!!!
- When a spinlock is used to protect data structures shared with interrupt handlers, the spinlock must disable interrupts
  - In this way, τ_i cannot be interrupted when it holds SL!

Spinlocks in Linux

- Defining a spinlock: `spinlock_t my_lock;`
- Initialising a spinlock: `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);`
  - `spin_unlock_bh(&my_lock);`
  - `spin_unlock_irqrestore(&my_lock, flags);`

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 preemption counter
  - `spin_unlock()` decreases the preemption counter
  - When the preemption counter returns to 0, `spin_unlock()` calls `schedule()`
- 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

- We call *atomic context* a CPU context in which it is not possible to sleep, block the current task, or invoke the scheduler
  - Interrupt handlers
  - Scheduler code
  - Critical sections protected by spinlocks
  - . . .
- What to do if I need to call a possibly-blocking function from atomic context?
  - Don't do it!!!
    - Try using the non-blocking version of the function...
    - Defer the work, to execute it later in a proper context $\rightarrow$ workqueues

Workqueues - 1

- Allow to schedule the execution of a function in the future
- The function will execute in a task context
- Lower priority than interrupt handlers, higher priority than user processes
- Using a workqueue:
  - include `<linux/workqueue.h>`
  - Creating: `wq = create_workqueue(name)`
  - Declaring the work to be done: `DECLARE_WORK(work, function, data)` (or: `INIT_WORK()` + `PREPARE_WORK()`
  - Scheduling the work: `queue_work(wq, work)`
  - Destroying: `destroy_workqueue(wq)`

Workqueues - 2

- After some work is scheduled for execution on a workqueue, the function will be called (in the future) in the context of a kernel thread serving the workqueue
- It is possible to force the execution of a workqueue (and to wait for it) by using `flush_workqueue(wq)`
- It is possible to schedule some work to be executed *after a timeout* (`queue_delayed_work()`)
- It is possible to cancel the execution of some work (`cancel_delayed_work()`)
- After using it, a workqueue can be destroyed (`destroy_workqueue()`)