Kernel and Locking

Luca Abeni

`luca.abeni@santannapisa.it`

Monolithic Kernels

- Traditional Unix-like structure
- Protection: distinction between Kernel (running in KS) and User Applications (running in US)
- The kernel behaves as a single-threaded program
  - One single execution flow in KS at each time
  - Simplify consistency of internal kernel structures
- Execution enters the kernel in two ways:
  - Coming from upside (system calls)
  - Coming from below (hardware interrupts)

Single-Threaded Kernels

- Only one single execution flow (thread) can execute in the kernel
  - It is not possible to execute more than 1 system call at time
    - Non-preemptable system calls
    - In SMP systems, syscalls are critical sections (execute in mutual exclusion)
  - Interrupt handlers execute in the context of the interrupted task

Bottom Halves

- Interrupt handlers split in two parts
  - Short and fast ISR
  - “Soft IRQ handler”
- Soft IRQ hanlder: *deferred* handler
  - Traditionally known ass Bottom Half (BH)
  - AKA Deferred Procedure Call - DPC - in Windows
  - Linux: distinction between “traditional” BHs and Soft IRQ handlers

Synchronizing System Calls and BHs

- Synchronization with ISRs by disabling interrupts
- Synchronization with BHs: is almost automatic
  - BHs execute atomically (a BH cannot interrupt another BH)
  - BHs execute at the end of the system call, before invoking the scheduler for returning to US
- Easy synchronization, but large non-preemptable sections!
  - Achieved by reducing the kernel parallelism
  - Can be bad for real-time

Latency in Single-Threaded Kernels

- Kernels working in this way are often called *non-preemptable kernels*
- L^{np} is upper-bounded by the maximum amount of time spent in KS
  - Maximum system call length
  - Maximum amount of time spent serving interrupts

Multiprocessor Issues

- Monolithic kernels are single-threaded: how to run then on multiprocessor?
  - The kernel is a critical section: Big Kernel Lock protecting every system call
  - This solution does not scale well: a more fine-grained locking is needed!
- Tasks cannot block on these locks $\rightarrow$ not mutexes, but *spinlocks*!
  - Remember? When the CS is busy, a mutex blocks, a spinlock spins!
  - Busy waiting... Not that great idea...

Removing the Big Kernel Lock

- Big Kernel Lock $\rightarrow$ huge critical section for everyone
  - Bad for real-time...
  - ...But also bad for troughput!
- Let's split it in multiple locks...
- Fine-grained locking allows more execution flows in the kernel simultaneously
  - More parallelism in the kernel...
  - ...But tasks executing in kernel mode are still non-preemptable

Spinlocks

- Spinlock: non-blocking synchronization object, similar to mutex
- Behave as a mutex, but tasks do not block on it
- A task trying to acquire an already locked spinlock spins until the spinlock is free
- Obviously, spinlocks are only useful on SMP
- For synchronising with ISR, there are “interrupt disabling” versions of the spinlock primitives
  - `spin_lock(lock)`, `spin_unlock(lock)`
  - `spin_lock_irq(l)`, `spin_unlock_irq(l)`
  - `spin_lock_irqsave(lock, flags)`, `spin_unlock_irqrestore(lock, flags)`

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

```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
```

Assembly 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!!!

Interrupt Handlers Context

- Remember: ISRs and BHs run in the context of the interrupted process
  - This is why they are in "Atomic Context" $\rightarrow$ cannot use mutexes
- What about giving them a proper context?
  - IRQ threads (hard - ISR - and soft - BH)
  - They are kernel threads activated when an interrupt fires
  - Proper context $\rightarrow$ can block, can use mutexes, ...
- When using IRQ threads, interrupt handler can be scheduled (like the other tasks)

IRQ Threads

- Supported (optionally) by Linux
- Kernel thread: thread which always execute in kernel mode
  - Created with `kthread_run()`
- Soft IRQ Threads and Hard IRQ Threads are just "regular" kernel threads...
  - Always blocked; become ready when a hardware interrupt (Hard IRQ) fires or a BH (Soft IRQ) is activated
  - Can use all of the kernel functionalities
  - A Hard IRQ Thread and a Soft IRQ Thread per IRQ

Task Descriptors

- On the Intel x86 architecture, TSS
  - It is a segment (described in GDT)
  - Current task descriptor $\leftarrow$ TR
- Stores the task context (CPU state, ...)
  - Stores EIP $\rightarrow$ task body
  - Stores CR3 $\rightarrow$ task address space
  - Stores ESP / SS $\rightarrow$ task stack (both US and KS)
- Used during context switches
- Can be linked in task queues

Context Switch

- Needed to multiplex many tasks on few CPUs
- 2 phases
  - Save the current CPU state in the task descriptor pointed by TR
  - Load the CPU state from a new task descriptor
- Can change CR3 (new address space)
  - Consequence: the CPU **MUST** have a high privilege level
  - Context switches happen in Kernel Space
- Changes the stack

Context Switches and Stack

- During a context switch, the kernel stack is changed...
- What about user stack???
    - Remember? The user-space ESP and SS are on the kernel stack...
- The User Space EIP and CS are on the stack too...
    - Returning to US, a different task will be executed...
- The context switch changes EIP too (kernel space)

The Scheduler

- A system has M CPUs $\rightarrow$ M tasks execute simultaneously
  - All the other tasks can be ready for execution or blocked
- Task descriptors are stored in queues
  - Ready task queue (can be global or per-CPU)
  - Blocked tasks queues
    - Condition variables
    - Mutexes and/or semaphores...
- the Scheduler selects tasks from the ready task queue

Task Queues

- There are multiple task queues inside the kernel
  - In the Linux kernel, they are implemented as lists (remember? `linux/list.h`)
- Tasks are inserted in the different queues according to their task state
  - Ready
  - Blocked on a completion / waitqueue / ...
  - ...
- Task states in Linux:
  - `TASK_RUNNING` → ready or executing
  - `TASK_INTERRUPTIBLE`, `TASK_UNINTERRUPTIBLE` → blocked

Blocking / Unblocking Tasks in Linux

- Tasks descriptors: `struct task_struct`
- Tasks are removed by the ready queue by the scheduler, when their state changes
  - Do not directly mess with the ready queue!
- How to block a task:
  - Change its state (`set_task_state()`)
  - Invoke the scheduler (`schedule()`)
- How to wake a task up:
  - `wake_up_process()`
- Note: sometimes, you can use higher-level abstractions (completions, waitqueues, ...)