

# Linux Kernel Architecture

## SMP issues, locking primitives

by

Harald Welte <laforge@gnumonks.org>

# Prerequisites

---

Due to the technical nature of this presentation, the audience should be familiar with the following subjects

- experience in programming on a Linux/\*NIX system
  - C language preferred
- general knowledge about computer hardware
  - interrupts / IO / DMA
- general knowledge about modern CPU architecture
  - address space / MMU
  - 'protected mode' / supervisor mode / ...

# Kernel / Userspace

---

- OS kernel provides
  - hardware abstraction (file I/O, network I/O, ...)
  - resource allocation / limiting
  - address separation
  - privilege separation
  - IPC
  
- the traditional process model in \*NIX operating systems
  - processes reside in separate virtual address spaces
  - kernel only executes one process (init) at bootup
  - all other processes descend from from init
  - processes are scheduled and preempted by the kernel
  - processes invoke system functions via syscalls.

# System calls

---

## Definition

- a userspace process enters the kernel
- mechanism is CPU architecture dependent
  - can be software interrupt (int 0x80)
  - can be special asm instruction (sysenter)
- arguments are passed on the stack
- common examples
  - open/close/read/write
  - exit/fork/execve/kill
  - socketcall, implements (socket/bind/connect/listen)
- about 270 system calls in 2.6.x kernels

# Invocation of system call

---

chronological order of events in case of a system call

- userspace process calls library function
  - library function is executed within the process' address space
  - library will eventually issue a syscall, pushing arguments on the stack
  - library will issue syscall (int 0x80 / sysenter / ...)
- execution will switch to syscall context in kernel mode
  - kernel will look up syscall table and dispatch to respective function
  - syscall function in the kernel will handle the syscall
  - all data between kernel/userspace needs to be copied between address spaces

# Execution contexts

---

apart from scheduling between different userspace processes, the kernel has different jobs like reacting to an external event

- hardirq**
  - hardware interrupt line was triggered
- softirq**
  - the workhorse behind a hardirq
- userspace**
  - executing within userspace process
- syscall**
  - invoked by a system call from userspace
- vsyscall**
  - virtual system calls, executed in userspace context

# hardirq context

---

- interrupt generated by hardware is received + handled
- can be interrupted by other hardirq's
- does only minimal job and returns
- examples
  - packet has arrived on network board
  - character was received on serial port
  - dma read/write to disk drive has completed
  - timer interrupt went off
  
- in most cases, a hardirq is followed by softirq or tasklet.

# softirq context

---

- softirqs are run after hardirq
- do the real work associated with a hardirq
- multithreaded (can run simultaneously on multiple cpus)
- examples
  - network receive softirq
  - timer softirq
  
- prior to softirq's, linux had so-called 'bottom halves'
  - softirq introduced in 2.4.x (net rx/tx softirq)
  - bottom halves removed in 2.6.x
  - difference: only one BH can be run at a time
  - BH's have to be converted to tasklets in 2.6.x

# tasklets

---

- tasklets are somewhat in between of softirq's and bottom halves
  - one particular tasklet cannot run on multiple CPUs simultaneously
  - different tasklets can run on different CPUs simultaneously
  
- otherwise, same as softirq context
  - tasklets are impl. inside the 'tasklet softirq'

# syscall / userspace context

---

- userspace context
  - in userspace, executing a process
  
- syscall context
  - inside kernel, when userspace process issues syscall()
  
- vsyscalls (virtual syscalls)
  - first introduced with the x86-64 (AMD Opteron) arch
  - fast read-only access to kernel data structures
  - can do stuff like `gettimeofday()` without context switch

# synchronization

---

Due to reentrancy and SMP, synchronization issues arise:

- simple case: UP system
  - softirq can be interrupted by hardirq
    - ▷ thus, shared structures (queues, ...) need to be protected
- complex case: SMP system
  - softirq can run at the same time on multiple CPU's
- as softirqs are multithreaded, synchronization between threads has to be implemented

# synchronization primitives

---

## busy-waiting locks

### □ spinlocks

- if lock was not taken, take it and continue
- if lock was taken, busy-loop until it is free

### □ rwlocks

- special case of spinlocks
- useful when structure protected by lock is often read but rarely updated/written to
- allows either
  - multiple readers simultaneously, or
  - only one writer [and no readers]

### □ brlocks

- super-fast read/write locks, with write-side penalty
- avoid cache ping-pong in multi reader case
- only in kernel 2.4.x

# synchronization primitives (cont'd)

---

## sleeper locks

### □ semaphores

- if semaphore can be acquired, continue
- if semaphore cannot be acquired, put current process to sleep
  - once semaphore is available again, wakeup process
  
- **WARNING:** can only be used for sync userspace/syscall context

# new locking primitives in 2.6.x

---

## □ seqlocks

- introduced with vsyscalls in 2.5/2.6
- reader/writer consistent mechanism without starving writers
- readers never block but may have to retry if write in progress

## □ read copy update

- new lockless mechanism in kernel 2.5/2.6
- defers update of data structure until all CPU's have scheduled and thus nobody has any references left

# example: incoming network packet

---

## hardirq context

- NIC issues interrupt line after a packet was received
- kernel enters (`arch/i386/kernel/entry.S:common_interrupt`)
- core interrupt handler (`arch/i386/kernel/irq.c:do_IRQ`)
- hardirq handler of network driver  
(`drivers/net/tulip/interrupt.c:tulip_interrupt`)
- `net/core/dev.c:netif_rx()`: append skb to backlog queue

# example: incoming network packet

---

## softirq context

- net/core/dev.c:net\_rx\_action()
- net/core/dev.c:process\_backlog()
- net/core/dev.c:netif\_receive\_skb()
- net/core/dev.c:deliver\_skb()
- net/ipv4/ip\_input.c:ip\_rcv()
  - netfilter prerouting hook
- net/ipv4/ip\_input.c:ip\_rcv\_finish()
  - call routing code
- net/ipv4/ip\_input.c:ip\_local\_deliver()
  - netfilter localin hook
- net/ipv4/ip\_input.c:ip\_local\_deliver\_finish()
  - call I4 protocol
- net/ipv4/udp.c:udp\_rcv()
  - lookup socket, if any
- include/net/sock.h:sock\_queue\_rcv\_skb()
  - enqueue into socket receiver queue
- net/core/sock.c:sock\_def\_readable()
  - wake\_up\_interruptible() on socket waitqueue
- return from recv() via socketcall

# Cache Effects

---

- ❑ SMP systems have multiple CPU's
- ❑ Every CPU has it's own cache(s) / cache hierarchies
- ❑ Most modern CPU archs are cache coherent in hardware
- ❑ This means a certain chunk of memory can only be write-cached on one CPU at a given time
  
- ❑ Frequently updated data structures will ping-pong between CPU caches
  
- ❑ Data structures have to be organized to avoid cache issues
  - Cacheline alignment
    - very easy by using `SLAB_HWCACHE_ALIGN`
  - per-cpu data structures
    - e.g. packet counters: have one for every CPU
  - structure layout
    - put all writeable/updated members together

# Thanks

---

□The slides and the an according paper of this presentation are available at <http://www.gnumonks.org/>

## □Thanks to

### ○the BBS people, Z-Netz, FIDO, ...

▷for heavily increasing my computer usage in 1992

### ○KNF

▷for bringing me in touch with the internet as early as 1994

▷for providing a playground for technical people

▷for telling me about the existance of Linux!

### ○Alan Cox, Alexey Kuznetsov, David Miller, Andi Kleen

▷for implementing (one of?) the world's best TCP/IP stacks

### ○Paul 'Rusty' Russell

▷for starting the netfilter/iptables project

▷for trusting me to maintain it today

### ○Astaro AG

▷for sponsoring parts of my netfilter work

### ○linux-bangalore

▷for sponsoring my trip to this conference