Linux Internals
Day 3
The Linux Kernel

Joel Isaacson
Ascender Technologies Ltd

System Calls

- The system call is (usually) called from user space.
- This causes a switch to kernel mode (ring 3 -> ring 0).
- It also causes the upper 1 giga byte (0xc0000000 – 0xffffffff) to be memory mapped.

System Calls

- The system call is a synchronous exception.
- It is actually either and “INT 80” or a “sysenter”.
- In Linux a new “vsyscalls” mechanism through the mapping to the kernel page “[vdso]”

System Calls

- Each Linux system call is given a number.
- The particular link to the called kernel routine is in the file “syscall_table.S”
- The xxxx system call is called sys_xxxx in the kernel.

Interrupts

- Interrupts allow hardware to communicate efficiently with the system software.
- The other possibility of communicating with hardware is via polling.
- Careful design of high speed communication devices has to deal with interrupt mitigation.

Interrupts

- Interrupt handlers are disruptive due to the fact that interrupts are disabled while they are active.
- Because of the interrupt handler's disruptive influence the interrupt handling is typically split into two parts:
  - Top half
  - Bottom half

Interrupts

- The top half typically does the minimal amount of work that has to be done until the interrupt is freed:
  - Copy any data
  - Re-enable the device
  - Acknowledge the interrupt

Interrupts

- The bottom half is run with interrupts enabled.
- This allows the computationally heavy part of the interrupt processing to be performed at a more convenient time.
- This allows interrupt latency to be low.

Interrupt Routine Registration

- include/linux/interrupt.h

```
#define IRQ_NONE        (0)
#define IRQ_HANDLED     (1)

// irqflags == SA_INTERRUPT | SA_SAMPLE_RANDOM | SA_SHIRQ

extern int request_irq(unsigned int irq,
   irqreturn_t (*handler)(int, void *, struct pt_regs *),
   unsigned long irqflags,
   const char *devname, void *dev_id);

extern void free_irq(unsigned int, void *);
```

Interrupt Context

- The interrupt routine runs in some random user context.
- The stack of the interrupt handler is:
  - Just the normal kernel stack for that random user context (8k stack kernels)
  - A per-cpu dedicated 4k interrupt stack (4k stack kernels)

Interrupt Context

- The interrupt handler's lack of its own context does not allow it to block.
- Thus many kernel interfaces are unusable in interrupt mode.
  - Memory allocation
  - Waiting for events
  - Timeouts

Shared Interrupts

- If all registered interrupts for a particular irq are registered SA_SHIRQ then multiple devices are handled on that particular irq.
- When an interrupt is indicated on that irq. All registered interrupt routines are called. They either return IRQ_NONE or IRQ_HANDLED.

/proc/interrupts

```
# cat /proc/interrupts
           CPU0       CPU1
  0:  151606351          0    IO-APIC-edge  timer
  1:     104737          0    IO-APIC-edge  i8042
  7:          0          0    IO-APIC-edge  parport0
  8:    8175643          0    IO-APIC-edge  rtc
  9:          1          0   IO-APIC-level  acpi
 12:    3770206          0    IO-APIC-edge  i8042
 14:    5433813          0    IO-APIC-edge  ide0
 50:          0          0   IO-APIC-level  uhci_hcd:usb2
 58:          0          0   IO-APIC-level  uhci_hcd:usb3
169:   82505113          0   IO-APIC-level  nvidia
177:   70410328          0   IO-APIC-level  uhci_hcd:usb4, eth0, HDA Intel
225:    3459023          0   IO-APIC-level  libata
233:    2538556          0   IO-APIC-level  uhci_hcd:usb1
NMI:      66696      68662
LOC:  151613505  151613482
ERR:          1
MIS:          0
```

Interrupt Control

- Disabling interrupts:
  - local_irq_disable()
  - local_irq_save(unsigned long flags)
  - disable_irq(unsigned int irq)
- Enabling interrupts:
  - local_irq_enable()
  - local_irq_restore(unsigned long flags)
  - enable_irq(unsigned int irq)

Non-Preemptive Kernels

- Unix kernels are “Non-Preemptive” traditionally.
- This refers solely to preemption in kernel mode.
- User mode is always preemptive.
- This means the scheduler can always run when the computer is in user mode.

Non-Preemptive Kernels

- Thus after an interrupt, during user mode, before the return to user mode the scheduler may be run causing a context switch.
- In kernel mode no context switch may be performed until the return to user mode.

Preemptive Kernels

- Linux in version 2.6 has gone to a “preemptive kernel”.
- This allows interrupts to preempt a process in kernel mode and switch to another process.
- The fact that this preemption can be done is based on the observation that a SMP-safe kernel is reentrant except for locked sections.

Preemptive Kernels

- Normally a request to preempt a process is caused by "`set_tsk_need_resched()`".
- This routine sets a flag that will cause rescheduling at the return from the interrupt routine:
  - in user mode
  - in kernel mode if the kernel was not holding any spinlocks.

Bottom Halves

- As we have seen interrupt processing is divided into two parts.
- The bottom half usually ends up doing the bulk the work.
- Time sensitive work is performed in the top half.
- Anything that deals with the hardware in in the top half.

Bottom Halves

- Linux has many bottom half mechanisms:
  - BH (< 2.5)
  - Task Queues (< 2.5)
  - Softirqs (> 2.3)
  - Tasklets (> 2.3)
  - Work Queues (> 2.5)

Softirqs

- A statically defined 32 entry array of routines.
- There is no mechanism of dynamically adding new types of softirqs.
- No preemptions among softirqs, except for interrupts.
- Multiple instances may run on different CPUs.

Softirqs

- Usually an interrupt handler will “mark” a softirq by setting a bit in the softirq mask.
- The priority is set by the location of the bit in the mask.
- This is the most low-level, efficient, difficult to use bottom half mechanism.

Softirqs

- There is a mechanism to prevent livelock.
- There is a “kernel thread” called softirqd that runs as a low priority scheduled “process” that performs the softirq work when the CPU begins to be inundated with work.

Kernel Threads

- Kernel threads have nothing to do with what is commonly considered threads.
- They are simply “user processes” without “user mode”.
- They have a process id, blockable context, they are scheduled but have no user memory space (addresses 0-3 Gb)

Softirqs

- The standard softirqs are:
  - HI_SOFTIRQ (high priority tasklets)
  - TIMER_SOFTIRQ (timer bottom half)
  - NET_TX_SOFTIRQ (network send)
  - NET_RX_SOFTIRQ (network receive)
  - SCSI_SOFTIRQ (scsi bottom half)
  - TASKLET_SOFTIRQ (tasklet normal prio)

Softirqs

- Softirqs are registered:
  - open_softirq(NET_TX_SOFTIRQ, action, 0)
- Softirqs are raised:
  - raised_softirq(NET_TX_SOFTIRQ)
- Softirqs are:
  - Hard to use since they may be running concurrently in many CPU's.
  - Very scalable for the same reason.

Tasklets

- Tasklets are softirqs with a human face.
- You usually want to use tasklets.
- They are easier to use.
- The locking rules for tasklets are simpler.
- Softirqs are for very frequent and/or heavily threaded uses.

Tasklets

- Tasklets are implemented with softirqs
- There are two classes of tasklets:
  - High priority, built on the HI_SOFTIRQ
  - Lower priority, built on TASKLET_SOFTIRQ
- Tasklets have the nice property that only one tasklet of a given type can run at any particular time.

Tasklets

- Tasklets can be dynamically created:
  - tasklet_init(struct tasklet_struct *t, func, data)
- Tasklets can be scheduled:
  - tasklet_schedule(struct tasklet_struct *t)
  - tasklet_hi_schedule(struct tasklet_struct *t)
- Since tasklets are based on softirqs on high load they are migrated to the softirqd kernel thread.

Work Queues

- Work queues are a different form of deferring work.
- They are always implemented as kernel threads.
- There have a process context and can block.
- They are higher level than tasklets. In general heavier

Work Queues

- They are used when the bottom half work might block.
  - Allocating memory
  - Acquiring semaphores
  - Reading/Writing files.
  - Sending/Receiving packets.

Work Queues

- Processor intensive and long blocking processing should be allocated application specific work queues so as not to impact the general (events/n) work queue.
- Work queue are scheduled and compete directly and fairly with user processes for CPU resources.

Work Queues

- Each work queue can have a “nice” parameter to “tweek” the total CPU time and responsiveness of particular work queues.
- The fact the work queues are scheduled prevent problems of livelock.

Work Queues

- Each work queue allocated one kernel thread per CPU.
- There is already created work default work queue that is used by many common users. The kthreads that perform this work queue's processing is called “events/*n*”, where *n* is the CPU number.

Work Queues

- Creating work queues:
  - `INIT_WORK(&work, void(*f)(void*), void *data);`
- Scheduling work:
  - `schedule_work(&work);`
- Scheduling delayed work:
  - `schedule_delayed_work(&work, delay);`
- Creating private work queue:
  - `pwork= create_workqueue(char *name);`

What to use? Softirqs

- Least serialization.
- Fastest.
- Handles livelock.
- Need per-CPU synchronization.
- Static allocations of enum priority at compile time.

What to use?
Tasklets

- Serialized.
- Efficient.
- Handles Livelock.
- No need for per-CPU variables.
- Easier that softirqs to write code.

What to use?
Work Queues

- Simple interface.
- Runs in process context.
- Scheduling is rather heavy.
- Can block.
- Not serialized.

What to use?
Kernel Threads

- Runs in process context.
- Scheduling is rather heavy.
- Can block.
- Not serialized.
- Easy to keep application context.

Kernel Synchronization

- Need to make sure that shared resources are protected from concurrent access.
- The kernel acts very much like a multi-thread application.
- Uncontrolled concurrent access to shared data will result in data corruption

Kernel Synchronization

- This is relatively easy with one processor
  - Just turn off interrupts an you know that the data isn't going anywhere.
- With SMP you might know that your CPU isn't messing with the data, but what about the other CPU's?
- With SMP you have to do better.

Critical Regions

- Even simple operation like i++ are not immune to data corruption.
- These effects are at times very subtle, difficult, if not impossible, to duplicate.
- This is what make kernel programming so expensive per line.

Race Conditions

- A typical case is:
  - if(cond == 0) do_a(); else do_b();
- Not trivial to fix this problem in all cases.

Locking

```
get_lock();
access_queue();
put_lock();
```

```
get_lock();
failed;
waiting;
succeeded;
access_queue();
put_lock();
```

Causes of Concurrency

- Interrupts.
- Softirqs and tasklets.
- Kernel preemption.
- Blocking and sleeping.
- SMP

What needs Protections

- Let's first figure out what doesn't need protection. It's simpler.

What doesn't need Protections

- Local data.
- Data that can be accessed by one task only (single threaded user applications).
- Private data in non-preemptive non-concurrent applications (tasklets).
- Private per CPU data in non-preemptive concurrent applications (softirqs).

What needs Protections?

- Just about everything else has to be examined carefully.