Unterbrechungen

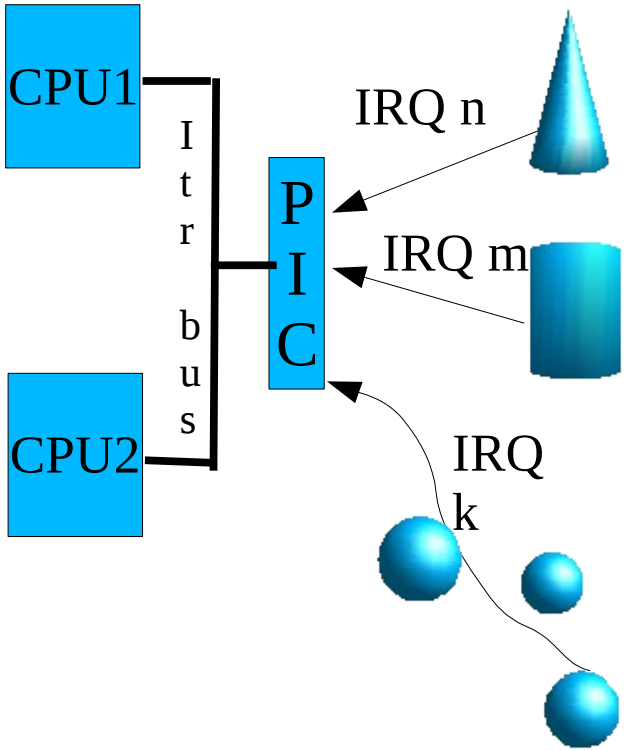
Reinhard Bündgen
buendgen@de.ibm.com

Was sind Unterbrechungen?

- Ereignisse, die den Instruktionsfluss auf einer CPU (temporär) unterbrechen
- oft asynchron zum Instruktionsfluss einer CPU
- Unterbrechungsquellen
  - Periphere Geräte
  - Zeitgeber des Computers
  - Prozessorausnahmen (Fehler, Traps)
  - Systemrufe
- Unterbrechungen können maskiert werden

Unterbrechungsarchitektur

- PCs
  - Geräte senden Interrupt Requests (IRQs) an Unterbrechungscontroller (PIC)
  - Unterbrechungscontroller sendet Interrupt (z.B. 8Bit Vektor) an CPUs
- zSeries Architektur
  - 64k channel subsystems
- einige IRQs sind fest zugeordnet
  - z.B auf PC ist IRQ 0 der timer interrupt



Genereller Unterbrechungsablauf

- analog zum Systemruf
  - alter program counter (PC) & CPU Status wird gerettet
  - neuer PC zeigt auf Unterbrechungsbehandlungsroutine & CPU ist im privilegiertem Modus
  - IRQ spezifische Routine wird über IRQ Tabelle berechnet
  - Während Unterbrechung können weitere Unterbrechungen maskiert werden.
    - Es werden in Linux üblicherweise auf allen Prozessoren Unterbrechungen des gleichen Typs maskiert
    - Unterbrechungen können von Unterbrechungen eines anderen Typs unterbrochen werden
  - Behandle Unterbrechung
  - Nach der Unterbrechungsbehandlung wird alter PC geladen und unterbrochener Code fortgeführt

Anforderungen an die Unterbrechungsbehandlung

- Bedingungen
  - Beachte Priorität von Unterbrechungen
  - Verhindere unendliche Verschachtelung von Unterbrechungen
  - Bediene alle Unterbrechungen
- Konsequenz
  - kurz & schnell
  - Maskiere Unterbrechungen von gleicher und niederer Priorität
  - Erledige aufwendige Arbeiten später:
    - top half & bottom half

Code Anforderungen und ISR

- Nicht im Task Context
- Interrupt Stack
  - kleine Task Stacks: Interrupt Stack pro Prozessor
    - `hardirq_stack` & `softirq_stack` Arrays (Intel)
  - große Task Stacks: benutzt Stack der unterbrochenen Task
- Top Half
  - darf nicht blockieren
  - muss nicht reentrant sei
- Bottom Half
  - darf nicht blockieren
  - läuft ohne Unterbrechungsmaskierung

Interrupt Handler Registierung

```
int request_irq(unsigned int irq,                irqreturn_t
  (*handler)(int, void *), unsigned long irqflags, const
  char *devname, void *dev_id)
```

- `irq`: interrupt number, static or dynamically allocated
- `irqreturn_t`: `IRQ_NONE` oder `IRQ_HANDLED`
- `irqflags`:
  - `IRQF_DISABLED` (alle Interrupts auf lokaler CPU disabled)
  - `IRQF_SAMPLE_RANDOM` (trägt zum Entropiepool bei)
  - `IRQF_SHARED` (IRQ gehört zu mehreren Geräten)
  - ...
- `*devname`
  - cf `/proc/irq` und `/proc/interrupts`
- `*dev_id`
  - beschreibt Gerät

Interrupt Handler

- Freigabe eines Interrupt Handlers

```
void free_irq(unsigned int irq, void *dev_id)
```

- Shared IRQs
  - `IRQF_SHARED` flag muss gesetzt sein
  - `*dev_id` muss eindeutig sein
  - Handler muss entscheiden können, ob sein Gerät den Interrupt erzeugt hat
  - Kern probiert alle Handler durch bis unterbrechendes Gerät gefunden
- Beispiel: real time clock
  - `drivers/char/rtc.c`
  - Registrierung in `rtc_init()`

Interrupt Behandlung

- `arch/`*`architecture`*`/kernel/entry.S`
- `do_IRQ` (z.B. `arch/i386/kernel/irq.c`)
  - stellt sicher, dass ein valider Handler registriert und ausführbar (läuft nicht)
- `handle_IRQ_event` (`kernel/irq/handle.c`)
  - läuft durch alle Handler für IRQ
- `ret_from_intr()`
  - tested ob `need_resched` gesetzt

Unterbrechungssteuerung

- `<asm/system.h>` und `<asm/irq.h>`
- Ein- und Ausschalten von Unterbrechungen für lokale CPU
  - alle: `local_irq_disable()` & `local_irq_enable()`
  - alle nicht maskierten: `local_irq_save(flags)` & `local_irq_restore(flags)`
  - Maskieren einer spezifischen Unterbrechung auf allen CPUs
  - `disable_irq(irq)` / `disable_irq_nosync(irq)`
  - `enable_irq(irq)` / `synchronize_irq(irq)`
- Status
  - in top oder bottom half: `in_interupt()`
  - in interrupt handler: `in_irq()`

Interprozessor Interrupts

- Viele SMP Systeme erlauben einer CPU anderen CPUs Interrupt Signale zu senden.
- Linux (i386) unterscheidet 3 Arten von Interprozessor Interrupts:
  - call function
    - Adressat soll Handler Funktion ausführen
  - reschedule
    - NOP, aber ret_from_interupt macht reschedule
  - invalidate TLB
    - Adressat invalidiert einen Teil seines TLB

Bottom Halves

- Letzter Teil einer Unterbrechungsbehandlung, der asynchron abgearbeitet werden kann
- Aufgabe der Top Half (Interrupt Handler)
  - Empfang der Unterbrechung quittieren
  - wichtige Daten von HW kopieren
  - IRQ freigeben
  - Evtl. Bottom Half aufsetzen
- Bottom Halves laufen mit unmaskierten IRQs

Bottom Half Mechanismen

Typ	**Verfügbar**	**Allokation**	**Context**	**Serialisierung**
BH	Bis 2.5			
Task queues	Bis 2.5			
Softirq	Seit 2.3	Zur Compile-Zeit	Interrupt	Keine
Tasklet	Seit 2.3	Zur Compile- oder Laufzeit	Interrupt	Gegen gleiches Tasklet
Work queues	Seit 2.5	Laufzeit	Task	Keine

Softirqs

- `softirq_action` Struktur in `<linux/interrupt.h>`
- `kernel/softirq.c`:
  - `static struct softirq_action softirq_vec[32];`
  - Bis jetzt nur 6 Einträge genutzt
- Können nur von Interrupts unterbrochen werden (nicht preemptable durch andere Softirqs)
- Können parallel auf verschiedenen CPUs laufen
- Dürfen nicht schlafen
- Anschalten von Softirqs (durch top half)
  - `void raise_softirq(unsigned int nr)`

Softirqs Ausführen

- Wann?
  - Nach HW Unterbrechungen
  - Durch ksoftirqd kernel thread (1 Thread pro CPU)
  - Wenn immer explizit programmiert (Netzwerk)
- Wo?
  - `do_softirq()` in `kernel/softirq.c`
- Wie?
  - Teste local_`softirq_pending` Bitvektor
  - Starte für alle angeschalteten softirqs die softirq-Handler

Definierte Softirqs

Name	Priorität	Beschreibung
HI_SOFTIRQ	0	Tasklets mit hoher Priorität
TIMER _SOFTIRQ	1	Timer bottom half
NET_TX_SOFTIRQ	2	Bottom half zum Senden von Netzwerk Paketen
NET_RX_SOFTIRQ	3	Bottom half zum Empfangen von Netzwerk Paketen
BLOCK_SOFTIRQ	4	block layer bottom half
TASKLET_SOFTIRQ	5	Tasklets mit normaler Priorität

Tasklets

- Sind keine kleinen Tasks!
- Sind als softirqs implementiert.
- 2 Prioritäten: `HI_SOFTIRQ` & `TASKLET_SOFTIRQ`
- Verschiedene Tasklets können auf verschieden CPUs parallel laufen, aber die gleiche Tasklet kann nur einmal laufen
- Taskletbeschreibung`: tasklet_sturct` Struktur
- Tasklet anschalten: `tasklet_schedule()`
- Tasklet Deklaration
  - statisch: `DECLARE_TASKLET(name,func,data)`
  - dynamisch:

```
void tasklet_init(struct tasklet_struct *t,
                  void (*handler)(unsigned long),
                  unsigned long data);
```

Tasklet Implementierung

Aus <linux/interrupt.h>:

```
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;             /* reference counter */
    void (*func)(unsigned long);          /* handler */
    unsigned long data;       /* argument to handler */
};
```

state: TASKLET_STATE_SCHED oder TASKLET_STATE_RUN (SMP only)

Tasklet Ausführen

- „angeschaltet“ (raised): scheduled
  - 2 per CPU Listen:
    - `tasklet_vec` für `TASKLET_SOFTIRQ`
    - `tasklet_hi_vec` für `HI_SOFTIRQ`
- Anschalten (raise): schedule
  - see `kernel/softirq.c, <linux/interrupt.h>`
  - `tasklet_schedule()`
  - `tasklet_hi_schedule()`
- Ausführen (via `do_softirq`)
  - `tasklet_action` Handler
  - `tasklet_hi_action` Handler

Arbeitsschlangen: Work Queues

- Option um Arbeit asynchron zu „starten“
- Läuft im Task-Kontext
- Darf schlafen, ist „schedulable“
- Laufen via worker kernel threads:
  - `worker_thread()` Funktion
  - Verschiedene worker thread Typen möglich
  - Ein thread pro CPU und worker thread Typ
  - Default Typ: `events/n`

Ablauf von Work Queues

- Siehe `kernel/workqueue.c`
- `worker_thread()`
  - Endlossschleife
  - Schläft, wenn nichts zu tun ist
  - Sonst ruft `run_workqueue()` auf
- `run_workqueue(struct cpu_workqueue_struct)`
  - Schleife über Liste von `work_struct` Elementen
  - Entnehme `work_struct` Funktion und Argument
  - Lösche `work_struct` von Liste
  - Rufe Funktion auf

Work Queues: Datenstrukturen

- kernel/workqueue.c:

  pro Typ

```
struct workqueue_struct {
    struct cpu_workqueue_struct cpu_wq[NR_CPUS];
    ...
};
```

  pro Typ pro CPU

```
struct cpu_workqueue_struct {
     ...
     struct list_head worklist; /* work */
     ...
     struct workqueue_struct *wq;
     ...
}
```

- **Work** <linux/workqueue.h>: struct work_struct

Neue Work (Queues)

- Erzeugen einer neuen Work Queue vom Typ `name:`
  - `struct workqueue_struct *create_workqueue(const char *name);`
- Arbeitsauftrag erzeugen
  - statisch: `DECLARE_WORK(name, void (*func)(void *),void *data);`
  - dynamisch: `INIT_WORK(struct work_struct *work, void (*func)(void *), void *data);`
  - `int queue_work(struct workqueue_struct *wq, struct work_struct *work);`
  - `int queue_delayed_work(struct workqueue_struct *wq, struct work_struct *work, unsigned long delay);`
- Alle Arbeitsaufträge löschen
  - `void flush_workqueue(struct workqueue_struct *wq);`
- Default Work Queue events:
  - `schedule_{delayed}_work(), flush_scheduled_work(), cancel_delayed_work()`