Zeit, Zeitmessung, Verzögerungen

Reinhard Bündgen
buendgen@de.ibm.com

Schlagworte

- Tick, HZ, jiffies, bogo mips
- relative vs. absolute Zeit
- Genauigkeit der Zeitmessung
- HW Uhren und Wecker
- Timer Interrupt & Timer ISR
- Dynamische Timer
- Warten
- Periodische Arbeiten

Periodische Aufgaben

- Aktualisieren der System uptime
- Aktualisieren der realen Zeit (TOD)
- Balancieren der Scheduler run queues in SMP Systemen
- Teste, ob current task seine Zeitscheibe verbraucht hat
- Starte timer Funktionen von abgelaufenen timern
- aktualisiere Statistiken über Ressource (e.g. CPU) Nutzung
- ....

Ticks

- Tick: Zeit zwischen 2 Timer Interrupts
  - Tickrate: HZ pro Sekunde
    - meist 100,
    - i386: 1000,
    - andere Werte: 24, 122, 32, 50, 1024
- System uptime
  - Anzahl der Ticks seit Bootzeitpunkt * HZ
- Wall clock time
  - TOD bei Boot + System uptime

Der Beste HZ Wert

- So groß wie möglich:
  - Höchste Präzision der Zeitmessung
  - Keine zu großen Perioden
    - Alle Perioden sind im Durchschnitt 0.5HZ zu lang
    - Wartezeiten, Zeitscheiben, ...
  - Task preemption
  - Präzisere Statistiken
- So klein wie möglich
  - 1 Timer Interrupt alle 1/HZ Sekunden
  - Timer Overhead
    - Instruktionen um Timer ISR auszuführen
    - Cash Thrashing
- Tickloses BS
  - Relative Zeit?
  - Verwalte dynamische Timer?

Jiffies

- 1 jiffy = 1s / `HZ`
- <linux/jiffies.h>
- `extern unsigned long volatile jiffies;`
- `jiffies` vs `jiffies_64`
- Jiffies wrap around
  - Nach 50 Tagen bei `HZ` = 100
  - Nach 497 Tagen bei `HZ` = 1000

Jiffies Wraparound

```
unsigned long timeout = jiffies + HZ/2;

...

if (time_after(jiffies, timeout)) {

    /* we did not time out */

} else

    /* we did time out */

}
```

```
#define time_after(unknown,known) ((long) (known) –
  (long) (unknown) < 0)
```

gibt true zurück wenn `unknown` später als `known` ist

HW zum Zeitnehmen

- Echtzeituhr (RTC)
  - Uhr die Uhrzeit (wall clock time) misst
  - meist batteriegetrieben, geht weiter auch wenn Computer aus ist
- System Timer
  - Eieruhr (Kurzzeitwecker)
  - Erzeugt periodische Unterbrechungen
  - Implementierung über CPU Frequenz oder Dekrementierfunktion
  - i386: programmable interrupt timer (PIT)
  - Wird beim Booten vom Kern initialisiert
- Problem
  - Zeitmessung auf virtuellen Systemen

Tageszeit

- `struct timespec xtime;`
  - Definiert in `kernel/timer.c`
  - `<linux/time.h>`

```
struct timespec {
        time_t tv_sec;      /* seconds */
        long tv_nsec;        /* nanoseconds */
}
```

  - Epoch: Zeit seit 1. Januar 1970
  - `xtime_lock`
- Userspace Funktion `gettimeofday()`
  - `sys_gettimeofday()`
    - `do_gettimeofday()` - uses sequence lock

Die Timer ISR

- Architektur abhängig
  - Reserviere `xtime_lock`
    - Beschützt `jiffies_64` und `xtime`
  - Setze System Timer zurück
  - Sichere xtime in die RTC (periodisch)
  - Rufe `do_timer` auf
- Architektur unabhängig (`kernel/timer.c`, kernel 2.6.9)
  - `do_timer` -> `update_process_times(); update_times()`
    - Inkrementiere `jiffies_64`
    - Aktualisiere Ressourcenutzungszeiten
    - Starte abgelaufene dynamische Timer (`run_local_timers()`)
    - `schedule_tick()`: dekrementiere Zeitscheibe von `current`, setzt evtl. `need_resched`, balanciert CUP run-queues
    - Aktualisiere `xtimer`
    - Berechne Durchschnittslast

Kern-Timers

- Erledige Kernel Arbeit später
  - aber nicht früher als angegebene Zeit
  - vgl. mit work queues: irgendwann später
- Nicht zyklisch, dynamisch: erzeugen und vernichten
- <linux/timer.h>:

```
struct timer_list {
  struct list_head entry;
  unsigned long expires; /* expiration in jiffies */
  /* spinlock_t lock; no longer in recent kernels */
  unsigned long magic;
  void (*function)(unsigned long);      /* handler*/
  unsigned long data;              /* arg to handle */
  struct tvec_t_base_s base;    /* internal field */
};
```

Kern Timer benutzten

```
struct timer_list my_timer;
init_timer(&my_timer);
my_timer.expires = jiffies+delay;  /* expires in delay ticks */
my_timer.data = 0;
my_timer.function = my_function;
add_timer(&my_timer);
=====
mod_timer(&my_timer, jiffies + new_delay);
del_timer(&my_timer);
del_timer_sync(my_timer);/* wait until executing timer exits */
```

Datenstruktur für dynamische Timer

•Ziel

- schnelle Operationen für das Einfügen, Löschen und Ablauferkennung.
- Annahme die wenigsten Timer laufen ab.

Lösung

eine Struktur pro CPU:

```
struct tvec_t_base_s {
    struct timer_base_s t_base;
    unsigned long timer_jiffies; /* earliest expiration */
    tvec_root_t tv1;       /* 1st 255 buckets            */
    tvec_t tv2;            /* 64 buckets: < 2^14 – 1     */
    tvec_t tv3;            /* 64 buckets: < 2^20 – 1     */
    tvec_t tv4;            /* 64 buckets: < 2^26 – 1     */
    tvec_t tv5;            /* 64 buckets: >= 2^26        */
} ____cacheline_aligned_in_smp;
```

Warten

- Aktives Warten (Busy Looping)
  - `while (time_before(jiffies,delay)) ;`
  - `while (time_before(jiffies,delay)) cond_resched();`
- Kurze Wartezeiten
  - `void udelay (unsignd long usecs);`
  - `void mdelay (unsinged long msecs);`
- Langes Warten
  - kernel/timer.c: `schedule_timeout()`
  - Erzeugt Timer mit Handler `process_timeout()` und Argument `current`
  - `process_timeout()` weckt current auf