Timekeeping in the Linux Kernel

Stephen Boyd
Qualcomm Innovation Center, Inc.
In the beginning ...
there was a counter

0000ec544fef3caf
Calculating the Passage of Time (in ns)

\[
\frac{c_{\text{cycles}}}{f_{\text{Hz}}} = \frac{c_{\text{cycles}}}{f\left(\frac{1}{\text{seconds}}\right)} = \left(\frac{c_{\text{cycles}}}{f}\right)_{\text{seconds}}
\]

\[
\left(\frac{c_{\text{cycles}}}{f}\right)_{\text{seconds}} = \frac{c_{\text{cycles}}}{f} \cdot 1e9 = T_{ns}
\]
Calculating the Passage of Time (in ns)

\[
\frac{c_{cycles}}{f_{Hz}} = \frac{c_{cycles}}{f\left(\frac{1}{seconds}\right)} = \left(\frac{c_{cycles}}{f}\right)seconds
\]

\[
\left(\frac{c_{cycles}}{f}\right)seconds = \frac{c_{cycles}}{f} \cdot 1e9 = T_{ns}
\]

Problems

- Division is slow
- Floating point math
- Precision/overflow/underflow problems
Calculating the Passage of Time (in ns) Better

```c
static inline s64 clocksource_cyc2ns(cycle_t cycles, u32 mult, u32 shift)
{
    return ((u64) cycles * mult) >> shift;
}
```
Calculating the Passage of Time (in ns) Better

```c
static inline s64 clocksource_cyc2ns(cycle_t cycles, u32 mult, u32 shift)
{
    return ((u64) cycles * mult) >> shift;
}
```

Where do mult and shift come from?

```c
clocks_calc_mult_shift(u32 *mult, u32 *shift, u32 from, u32 to, u32 minsec)
```
Abstract the Hardware!

```c
struct clocksource {
    cycle_t (*read)(struct clocksource *cs);
    cycle_t mask;
    u32 mult;
    u32 shift;
    ...
};

clocksource_register_hz(struct clocksource *cs, u32 hz);
clocksource_register_khz(struct clocksource *cs, u32 khz);
```

Time diff:

```c
struct clocksource *cs = &system_clocksource;
cycle_t start = cs->read(cs);
... /* do something for a while */
cycle_t end = cs->read(cs);
clocksource_cyc2ns(end - start, cs->mult, cs->shift);
```
POSIX Clocks

- CLOCK_BOOTTIME
- CLOCK_MONOTONIC
- CLOCK_MONOTONIC_RAW
- CLOCK_MONOTONIC_COARSE
- CLOCK_REALTIME
- CLOCK_REALTIME_COARSE
- CLOCK_TAI
POSIX Clocks Comparison

CLOCK_BOOTTIME

CLOCK_MONOTONIC

CLOCK_REALTIME
Read Accumulate Track (RAT)

*Best acronym ever*
struct tk_read_base {
    struct clocksource *clock;
    cycle_t (*read)(struct clocksource *cs);
    cycle_t mask;
    cycle_t cycle_last;
    u32 mult;
    u32 shift;
    u64 xtime_nsec;
    ktime_t base;
};

static inline u64 timekeeping_delta_to_ns(struct tk_read_base *tkr, cycle_t delta) {
    u64 nsec = delta * tkr->mult + tkr->xtime_nsec;
    return nsec >> tkr->shift;
}

static inline s64 timekeeping_get_ns(struct tk_read_base *tkr) {
    cycle_t delta = (tkr->read(tkr->clock) - tkr->cycle_last) & tkr->mask;
    return timekeeping_delta_to_ns(tkr, delta);
### RAT in Action (Accumulate + Track)

```c
static u64 logarithmic_accumulation(struct timekeeper *tk, u64 offset, u32 shift, unsigned int *clock_set)
{
    u64 interval = tk->cycle_interval << shift;
    tk->tkr_mono.cycle_last += interval;
    tk->tkr_mono.xtime_nsec += tk->xtime_interval << shift;
    *clock_set |= accumulate_nsecs_to_secs(tk);
    ...
}

static inline unsigned int accumulate_nsecs_to_secs(struct timekeeper *tk)
{
    u64 nsecps = (u64)NSEC_PER_SEC << tk->tkr_mono.shift;
    unsigned int clock_set = 0;
    while (tk->tkr_mono.xtime_nsec >= nsecps) {
        int leap;
        tk->tkr_mono.xtime_nsec -= nsecps;
        tk->xtime_sec++;
    }
    ...
}
```
Juggling Clocks

```c
struct timekeeper {
    struct tk_read_base   tkr_mono;
    struct tk_read_base   tkr_raw;
    u64                    xtime_sec;
    unsigned long          ktime_sec;
    struct timespec64      wall_to_monotonic;
    ktime_t                offs_real;
    ktime_t                offs_boot;
    ktime_t                offs_tai;
    s32                     tai_offset;
    struct timespec64      raw_time;
};
```
Handling Clock Drift

\[
\frac{1}{19200000} \cdot 1e9 = 52.083_{ns}
\]

Vs.

\[
\frac{1}{19200008} \cdot 1e9 = 52.083311_{ns}
\]
Handling Clock Drift

\[
\frac{100000}{19200000} \cdot 1e9 = 520833_{ns}
\]

Vs.

\[
\frac{100000}{19200008} \cdot 1e9 = 5208331_{ns}
\]

After 100k cycles we've lost 2 ns
Mult to the Rescue!

\[(100000 \cdot 873813333) \gg 24 = 5208333_{ns}\]

Vs.

\[(100000 \cdot 873813109) \gg 24 = 5208331_{ns}\]

Approach: Adjust mult to match actual clock frequency
Making Things Fast and Efficient

static struct {
    seqcount_t       seq;
    struct timekeeper timekeeper;
} tk_core  __cacheline_aligned;

static struct timekeeper shadow_timekeeper;

struct tk_fast {
    seqcount_t       seq;
    struct tk_read_base base[2];
};

static struct tk_fast tk_fast_mono  __cacheline_aligned;
static struct tk_fast tk_fast_raw  __cacheline_aligned;
A Note About NMIs and Time
Where We Are
What if my system doesn't have a counter?

Insert #sadface here

- Can't use NOHZ
- Can't use hrtimers in "high resolution" mode

Relegated to the jiffies clocksource:

```c
static cycle_t jiffies_read(struct clocksource *cs)
{
    return (cycle_t) jiffies;
}

static struct clocksource clocksource_jiffies = {
    .name           = "jiffies",
    .rating         = 1, /* lowest valid rating*/
    .read           = jiffies_read,
    ...
};
```
Let's talk about jiffies
Let's talk about jiffies

\[ \text{Jiffy} = \frac{1}{\text{CONFIG_HZ}} \]
Let's talk about jiffies

\[ \text{Jiffy} = \frac{1}{\text{CONFIG_HZ}} \]

Updated during the "tick"
The tick?
The tick

Periodic event that updates

- jiffies
- process accounting
- global load accounting
- timekeeping
- POSIX timers
- RCU callbacks
- hrtimers
- irq_work
Implementing the tick in hardware

Timer Value: 4efa4665
Match Value: 4efa4666
Abstract the Hardware!

```c
struct clock_event_device {
    void (*event_handler)(struct clock_event_device *);
    int (*set_next_event)(unsigned long evt,
                          struct clock_event_device *);
    int (*set_next_ktime)(ktime_t expires,
                          struct clock_event_device *);
    ktime_t next_event;
    u64 max_delta_ns;
    u64 min_delta_ns;
    u32 mult;
    u32 shift;
    unsigned int features;
    #define CLOCK_EVT_FEAT_PERIODIC 0x000001
    #define CLOCK_EVT_FEAT_ONESHOT 0x000002
    #define CLOCK_EVT_FEAT_KTIME 0x000004
    int irq;
    ...
};

void clockevents_config_and_register(struct clock_event_device *dev,
                                      u32 freq, unsigned long min_delta, 
                                      unsigned long max_delta)
```

Three event_handlers

```c
struct clock_event_device {
    void (*event_handler)(struct clock_event_device *);
    int (*set_next_event)(unsigned long evt, struct clock_event_device *);
    int (*set_next_ktime)(ktime_t expires, struct clock_event_device *);
    ktime_t next_event;
    u64 max_delta_ns;
} ...
```

<table>
<thead>
<tr>
<th>Handler</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>tick_handle_periodic()</td>
<td>default</td>
</tr>
<tr>
<td>tick_nohz_handler()</td>
<td>lowres mode</td>
</tr>
<tr>
<td>hrtimer_interrupt()</td>
<td>highres mode</td>
</tr>
</tbody>
</table>
Ticks During Idle

tick_handle_periodic()
Tick-less Idle (i.e. CONFIG_NOHZ_IDLE)

tick_handle_periodic()

```
t1
```

```
```

tick_nohz_handler()

```
t1
```

```
```

time

31 / 40
High Resolution Mode

tick_nohz_handler()

hrtimer_interrupt()
Tick Devices

```c
enum tick_device_mode {
    TICKDEV_MODE_PERIODIC,
    TICKDEV_MODE_ONESHOT,
};

struct tick_device {
    struct clock_event_device *evtdev;
    enum tick_device_mode mode;
};

DEFINE_PER_CPU(struct tick_device, tick_cpu_device);
```
Running the Tick

```c
struct tick_sched {
    struct hrtimer sched_timer;
    ...
};
```
Running the Tick (Per-CPU)

```c
struct tick_sched {
    struct hrtimer sched_timer;
    ...
};

DEFINE_PER_CPU(struct tick_sched, tick_cpu_sched);
```
# Stopping the Tick

- Not always as simple as
  ```
  hrtimer_cancel(&ts->sched_timer)
  ```

- Could be that we need to restart the timer so far in the future
  ```
  hrtimer_start(&ts->sched_timer, tick, HRTIMER_MODE_ABS_PINNED)
  ```

Needs to consider:

- timers
- hrtimers
- RCU callbacks
- jiffies update responsibility
- clocksource's max_idle_ns (timekeeping max deferment)

*Details in tick_nohz_stop_sched_tick()*
Tick Broadcast

- For when your clock events **FAIL AT LIFE**
  - i.e., they don't work during some CPU idle low power modes
  - Indicated by CLOCK_EVT_FEAT_C3_STOP flag
Timers

- Operates with jiffies granularity
- Requirements
  - jiffies increment
  - clockevent
  - softirq
HRTimers

- Operates with ktime (nanoseconds) granularity
- Requirements
  - timekeeping increment
  - clockevent
  - tick_device
## Summary

### What we covered
- clocksources
- timekeeping
- clockevents
- jiffies
- NOHZ
- tick broadcast
- timers
- hrtimers

### What’s difficult
- Timekeeping has to handle NTP and drift
- Tick uses multiple abstraction layers
- NOHZ gets complicated when starting/stoping the tick
- Tick broadcast turns up NOHZ to 11