The Ticking Beast

A deep dive into Timekeeper, Timers, Tick and Tickless kernels.

(Linux Foundation webinar: Feb 22nd 2024)

Joel Fernandes (Google)

joel@joelfernandes.org

Who am I ?

Joel Fernandes (Google)

joel@joelfernandes.org

Agenda

- Userspace **time** APIs
- Clocksource
  - Time stamp counter (TSC)

25 minutes

- Userspace **timer** APIs
- Clockevents
  - Local APIC timer
  - HPET
  - Broadcast timers

25 minutes

- Timer wheel
- Hrtimer
- Scheduling clock interrupt (tick) and NOHZ
- VDSO (if time permits)

40 minutes

Userspace

- How do you get the current time?
  - `clock_gettime()` API
    - `int clock_gettime(clockid_t clockid, struct timespec *tp);`

      ```
      struct timespec {
         time_t   tv_sec;        /* seconds */
         long     tv_nsec;       /* nanoseconds */
      };
      ```
  - Clock IDs for keep track of elapsed time.
    - `CLOCK_REALTIME`
    - `CLOCK_MONOTONIC`
    - `CLOCK_BOOTIME`
  - `gettimeofday()` directly operates on `CLOCK_REALTIME`.

Userspace

- Let us go over Clock IDs
  - `CLOCK_REALTIME`
    - affected by changes in time by user
    - NTP (adjtime).
      - Used to correct time by adjusting clock rate till time is corrected.

Userspace

- Let us go over Clock IDs
  - `CLOCK_MONOTONIC`
    - NOT affect by changes in time by user.
    - Affected by changes in time by adjtime (NTP changes clock rate).
    - Does NOT count suspend time.

Userspace

- Let us go over Clock IDs
  - `CLOCK_BOOTTIME`
    - Identical to `CLOCK_MONOTONIC` except..
    - Accounts for **suspend time.**

Userspace

- Clock ID behavior summary

Clock ID name	Time since	Can be set by user?	Can be set my adjtime	Accounts suspend time?
`CLOCK_REALTIME`	Epoch	Yes	Yes	Yes
`CLOCK_MONOTONIC`	Boot	No	Yes	No
`CLOCK_MONOTONIC_RAW`	Boot	No	No	No
`CLOCK_BOOTTIME`	Boot	No	Yes	Yes

Userspace

- How do you set the time?
  - `clock_settime()` - set the time of the specified clock clockid.
    - `int clock_settime(clockid_t clockid, const struct timespec *tp);`
  - `adjtime()` - gradually correct the time
    - `int adjtime(const struct timeval *delta, struct timeval *olddelta);`
    - Clock is sped up over slow down a bit every second.
    - Typically used by NTP to adjust for clock drift.
  - `settimeofday()` counterpart to `gettimeofday()`.

Userspace

- How to get resolution of a clock?

```
int clock_getres(clockid_t clockid, struct timespec *res);

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

Now lets look at how timekeeping is supported in the kernel..

Buckle up :)

Kernel support - timekeeping

- How does the kernel track different clocks?

```
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;
        unsigned int            clock_was_set_seq;
        u8                      cs_was_changed_seq;
        ktime_t                 next_leap_ktime;
        u64                     raw_sec;
        struct timespec64       monotonic_to_boot;
```

Time is accumulated here `CLOCK_MONOTONIC` and `CLOCK_MONOTONIC_RAW`

Offset to `CLOCK_REALTIME`

Offset to `CLOCK_BOOTTIME`

Offset to `CLOCK_TAI`

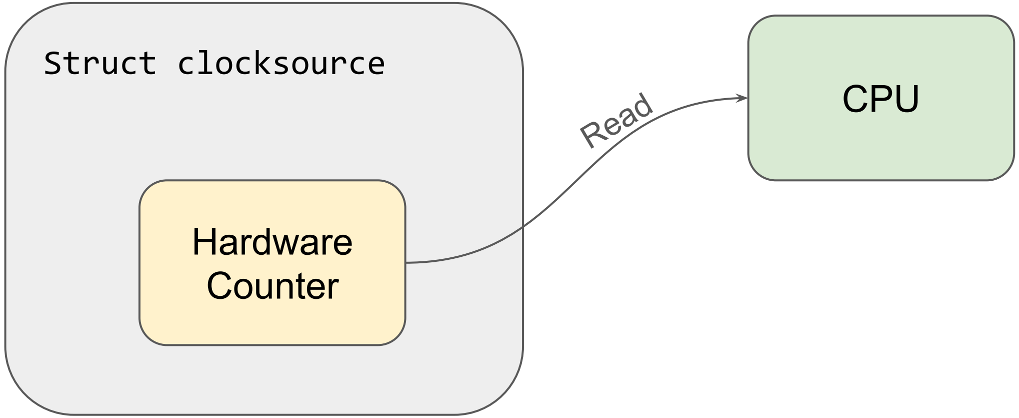
Kernel support - timekeeping

- Several timekeeping APIs are in VDSO.
- For instance, to get time userspace reads TSC and scales cycle delta from last read.

```
/**
 * struct vdso_data - vdso datapage representation
 * @seq:               timebase sequence counter
 * @clock_mode:        clock mode
 * @cycle_last:        timebase at clocksource init
 * @mask:              clocksource mask
 * @mult:              clocksource multiplier
 * @shift:             clocksource shift
 * @basetime[clock_id]: basetime per clock_id
 * @offset[clock_id]:  time namespace offset per clock_id
 * @tz_minuteswest:    minutes west of Greenwich
 * @tz_dsttime:        type of DST correction
 * @hrtimer_res:       hrtimer resolution
 * @__unused:          unused
 * @arch_data:         architecture specific data (optional, defaults
 *                     to an empty struct)
 *
 * vdso_data will be accessed by 64 bit and compat code at the same time
 * so we should be careful before modifying this structure.
 *
 * @basetime is used to store the base time for the system wide time getter
 * VVAR page.
 *
 * @offset is used by the special time namespace VVAR pages which are
```

Kernel Support - timekeeping - Clocksource

A clocksource is an abstraction on simple clock (counter) that can be read from!



Kernel Support - timekeeping - Clocksource

Example: x86 Time stamp counter (TSC)

- 64-bit per-CPU counter, it is an MSR so fast!!! (slower than cache hit!)
- High resolution (GHz), uses the CPU clock.
- Read using the `RDTSC` instruction
- `RDTSCP` also gives the CPU number on which the TSC was read.

Kernel Support - Clocksource: Abstraction of the hardware

- Clocksource kernel API

```
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 difference

```
// Note that this breaks if clocksource on all CPUs are not synced!
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);
```

So what do we use the clocksource for?

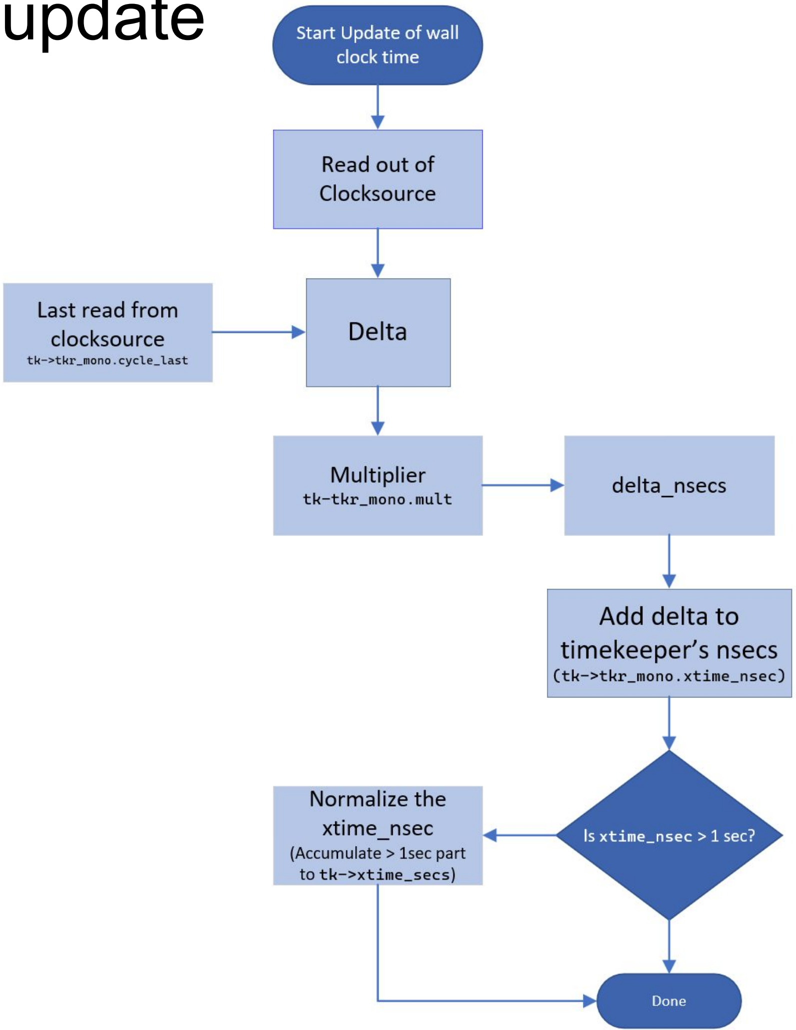
- Timekeeping: Moving time in the system forward.
- Reading time at a given instant.

Kernel Support - Timekeeper update

1. Clocksource read during `update_wall_time()`

`New clock = ((last_cycle - current cycle) * multiplier) + Old.`

2. Update is done every jiffy.



Kernel Support - Timekeeper update

To summarize previous chart.

- Clocksource is read and accumulated into `struct timekeeper:`
  - This structure has 2 components to keep track of time in seconds.
    - `xtime_nsec` : The time so far in nanoseconds.
    - `xtime_sec` : If the nsecs grows more than a second, it overflows into this element.
    - Number of cycles during last clocksource read is noted during every TK update.
      - Needed to update timekeeping.
      - As we'll see next, needed to read instantaneous time as well.

Kernel Support - Timekeeper readout

Q: That's every jiffy but.. How is time at any **instant** read?

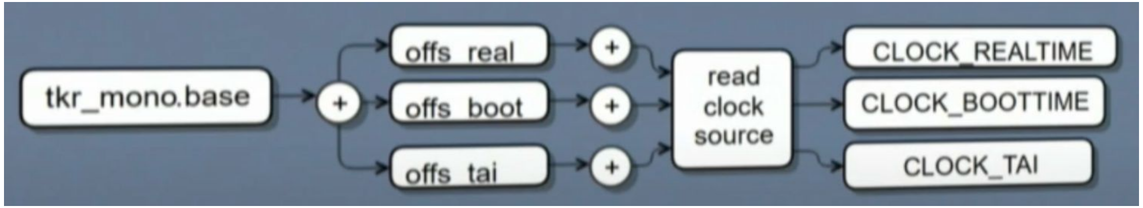
Ans: Timekeeper (last slide) + Clocksource Read (delta) + Adjustments

```
struct timekeeper {
    struct tk_read_base tkr_mono;
    u64          xtime_sec;
    ktime_t          offs_real;
    ktime_t          offs_boot;
    ktime_t          offs_tai;
    ...
}
```

Updated by `update_wall_time()`

Updated by NTP or `clock_settime()` for `CLOCK_REALTIME`

Updated with suspend time for `CLOCK_BOOTTIME`



Kernel Support - Timekeeping Accumulation (Code)

Few more things for completeness:

- Wallclock time is updated every jiffy by a designated CPU:
  - ```
    tick_nohz_highres_handler() ->
                tick_sched_do_timer() ->
                        tick_do_update_jiffies64() ->
                              update_wall_time()
    ```
    ```

Now let us jump into a real example of an x86 clock source

-- our old friend TSC again.

And what issues plague the TSC?

x86 Time stamp counter (TSC)

- 64-bit per-CPU counter, it is an MSR so fast!!! (slower than cache hit!)
- High resolution (GHz), uses the CPU clock.

Kernel Support - Clocksource - TSC issues

TSC stability (frequency invariance).

- CPU clock can change frequency and affect TSC increment rate.
- Older CPU models unreliable to frequency dep, but recently constant.
  - Check "`constant_tsc`" flag in `/proc/cpuinfo`
- If CPU does not have constant_tsc feature, then if cpufreq changes, TSC marked unstable (`mark_tsc_unstable()`).
- Clocksource reselection happens once TSC clocksource is marked unstable. Switches to HPET via clocksource watchdog kthread.

Kernel Support - Clocksource - TSC issues

TSC stoppage (due to deep idle)

- TSC can stop counting in idle states because depends on CPU clock liveness.
- CPU PM may effect
  - Check "`nonstop_tsc`" flag in `/proc/cpuinfo`
- If CPU does not have nonstop_tsc feature, then idle driver may mark TSC unstable (`mark_tsc_unstable()`) if deeper than C2 state is allowed / chosen.
- Clocksource reselection happens once TSC clocksource is marked unstable. Switches to HPET.

Kernel Support - Clocksource - catch it red handed

Clocksource watchdog to keep an eye on clocksource stability

- A timer is scheduled to run every half a second to verify clocksource stability for clocksources with `CLOCK_SOURCE_MUST_VERIFY` flag.
- Another clocksource that does not have `CLOCK_SOURCE_MUST_VERIFY` is compared against. If large difference between the 2 clocksource’s understanding of time progression, clocksource is marked unstable.
- Once marked unstable, kthread worker selects a new clocksource (like HPET for x86).

That’s it for clock source, timestamps..

Now lets see how timer events are handled

Userspace - Timers (will just skim through userspace to spend more time on the kernel part)

Timer is a mechanism to generate a notification at a future point of time.

- POSIX timers
- timerfd
- sleep
- timeouts for syscalls
- hrtimer user in kernel

Userspace - POSIX timers

```
int timer_create(clockid_t clockid, struct sigevent *sevp, timer_t *timerid);
```

- Create a **per-process interval** timer. Returns unique timer ID
- Clockid is any of the clocks we discussed.
  - Some additional special clocks exist such as:
    - `CLOCK_PROCESS_CPUTIME_ID` - measures CPU time consumed by all threads.
    - `CLOCK_THREAD_CPUTIME_ID` - same but just for calling thread.
- struct sigevent : specifies how the caller should be notified when the timer expires.

Userspace - POSIX timers - arming

```
int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *new_value,
                  struct itimerspec *old_value);
int timer_gettime(timer_t timerid, struct itimerspec *curr_value);
                returns the time until next expiration & the interval
struct itimerspec {
    struct timespec it_interval;  /* Timer interval, (If 0, then timer is ONESHOT) */
    struct timespec it_value;     /* Initial expiration (relative to current time, can be changed by flags)
                                     (If 0, disarms the timer) */
};
struct timespec {
  time_t tv_sec;                /* Seconds */
  long   tv_nsec;               /* Nanoseconds */
};
```

Userspace - POSIX timers - arming

```
int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *new_value,
                  struct itimerspec *old_value);
int timer_gettime(timer_t timerid, struct itimerspec *curr_value);


struct itimerspec {
    struct timespec it_interval;
    struct timespec it_value;
};
```

Provide new interval

Userspace - POSIX timers - arming

```
int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *new_value,
                  struct itimerspec *old_value);
int timer_gettime(timer_t timerid, struct itimerspec *curr_value);
```

Provide new interval

```
struct itimerspec {
    struct timespec it_interval;
    struct timespec it_value;
};
```

Initial expiration (relative to current time, can be changed by flags)

Userspace - POSIX timers - arming

```
int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *new_value,
                  struct itimerspec *old_value);
int timer_gettime(timer_t timerid, struct itimerspec *curr_value);
```

Provide new interval

```
struct itimerspec {
    struct timespec it_interval;
    struct timespec it_value;
};
```

Frequency of expiration, If zero, the timer is one shot.

Initial expiration (relative to current time, can be changed by flags)

Userspace - POSIX timers

What does the kernel do internally?

- For each clock, there is a `struct kclock`.
- As you can see, it uses hrtimer under the hood.

```
static const struct k_clock clock_realtime = {
	.clock_getres		= posix_get_hrtimer_res,
	.clock_get_timespec	= posix_get_realtime_timespec,
	.clock_get_ktime	= posix_get_realtime_ktime,
	.clock_set		= posix_clock_realtime_set,
	.clock_adj		= posix_clock_realtime_adj,
	.nsleep			= common_nsleep,
	.timer_create		= common_timer_create,
	.timer_set		= common_timer_set,
	.timer_get		= common_timer_get,
	.timer_del		= common_timer_del,
	.timer_rearm		= common_hrtimer_rearm,
	.timer_forward		= common_hrtimer_forward,
	.timer_remaining	= common_hrtimer_remaining,
	.timer_try_to_cancel	= common_hrtimer_try_to_cancel,
	.timer_wait_running	= common_timer_wait_running,
	.timer_arm		= common_hrtimer_arm,
};

static const struct k_clock clock_monotonic = {
	.clock_getres		= posix_get_hrtimer_res,
	.clock_get_timespec	= posix_get_monotonic_timespec,
	.clock_get_ktime	= posix_get_monotonic_ktime,
	.nsleep			= common_nsleep_timens,
	.timer_create		= common_timer_create,
	.timer_set		= common_timer_set,
	.timer_get		= common_timer_get,
	.timer_del		= common_timer_del,
	.timer_rearm		= common_hrtimer_rearm,
	.timer_forward		= common_hrtimer_forward,
	.timer_remaining	= common_hrtimer_remaining,
	.timer_try_to_cancel	= common_hrtimer_try_to_cancel,
	.timer_wait_running	= common_timer_wait_running,
	.timer_arm		= common_hrtimer_arm,
};
```

A note on alarm clock ids and POSIX timers

- There are 2 additional clock ids that can be used with userland timers:
  - `CLOCK_REALTIME_ALARM`
  - `CLOCK_BOOTTIME_ALARM`

- When used, they wake the system up even during suspend. See `kernel/time/alarmtimer.c`

- Uses RTC hardware which is active even when the system is suspended.

Userspace - timerfd

- File descriptor based timers
- Advantage is, can use select/poll because of fd.
- This also allows uses hrtimer under the hood.
- Will not go over more details, check documentation.

Userspace - Comparing POSIX timers and timerfd

Feature	timerfd	POSIX Timers
Identifier	File descriptor	Timer ID
Closing/Deletion	close() on the file descriptor	timer_delete()
Creation	timerfd_create()	timer_create()
Configuration/Arming	timerfd_settime()	timer_settime()
Portability	Linux-specific	POSIX standard, wider portability across Unix-like systems
Synchronization	Simplifies synchronization by using file descriptors	Requires careful signal handling, especially in multithreaded environments
Integration with Event Loops	Natural fit for event loops using epoll, select, or poll	Can we made to work with event loops but requires additional step like signalfd.

Kernel Support - Clockevents and timers

A clockevent device abstracts a device which generates interrupt at programmed time in the future.

There are 2 types of clockevents:

- Per-CPU -- dependent of CPU , example LAPIC timer.
- Global -- independent of CPU , example HPET.

Kernel Support - Clockevents

A clockevent device abstracts a device which generates interrupt at programmed time in the future.

```
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);
```

Kernel Support - Clockevents

A clockevent device abstracts a device which generates interrupt at programmed time in the future.

```
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);
```

Program next event (relative and absolute).

Kernel Support - Clockevents

A clockevent device abstracts a device which generates interrupt at programmed time in the future.

```
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);
```

Run callback on next event.

Kernel Support - Clockevents

A clockevent device abstracts a device which generates interrupt at programmed time in the future.

```
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);
```

Run callback on next event.

Clock event features. ONESHOT is required for NOHZ

Kernel Support - Clockevents

Clockevent drives the timer events on every CPU

```
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);
```

Timer wheel timers

HRTimer timers

Timekeeping, Periodic Tick

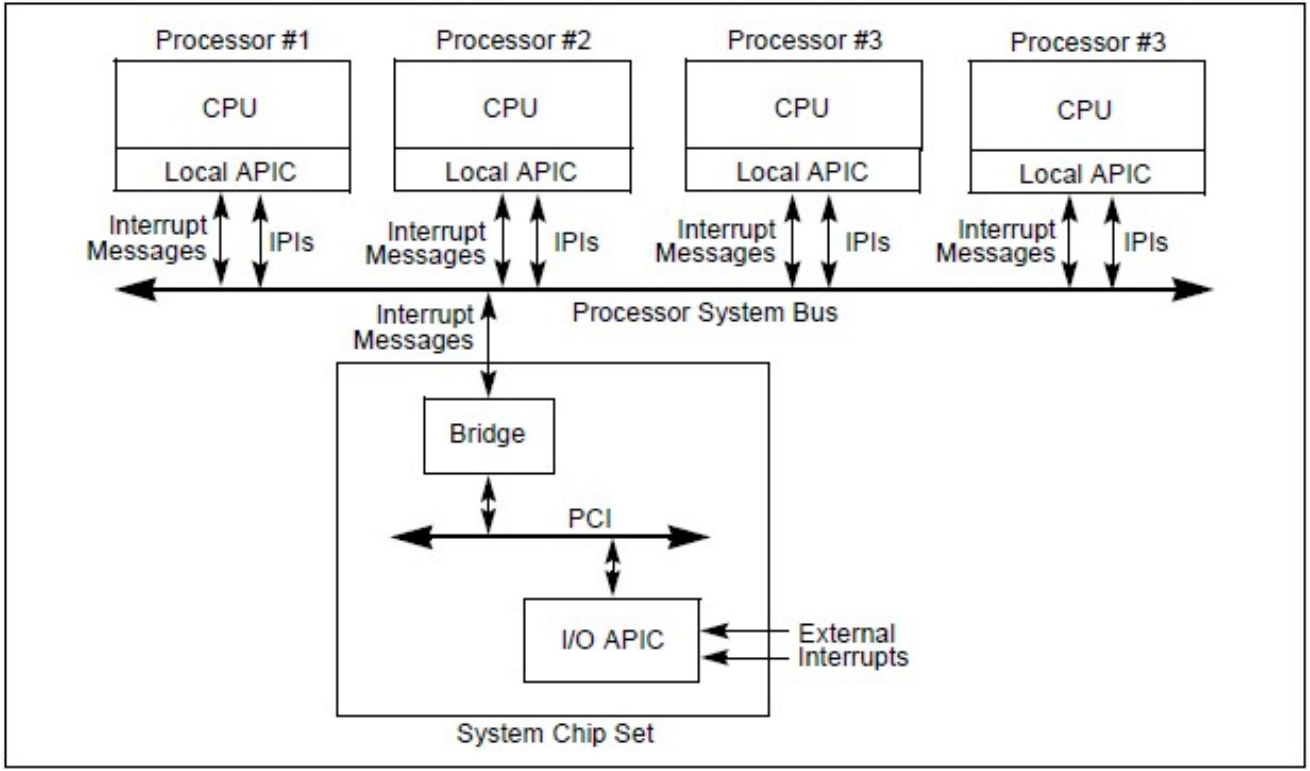
Kernel Support - Clockevents

Clockevent Example: Local APIC timer (lapic)

- Per-CPU Interrupt Controller with a timer.
- Tightly coupled with CPU core.
- Low precision (~MHz) as countdown rate determined by external bus freq.
- Has a “TSC deadline mode” which gives it GHz precision.
    - Generates an IRQ whenever TSC crosses certain value.
    - Write absolute TSC deadline to `IA32_TSC_DEADLINE` MSR arms it.

Kernel Support - Clockevent

Clockevent Example: Local APIC timer (lapic)



Kernel Support - Clockevent

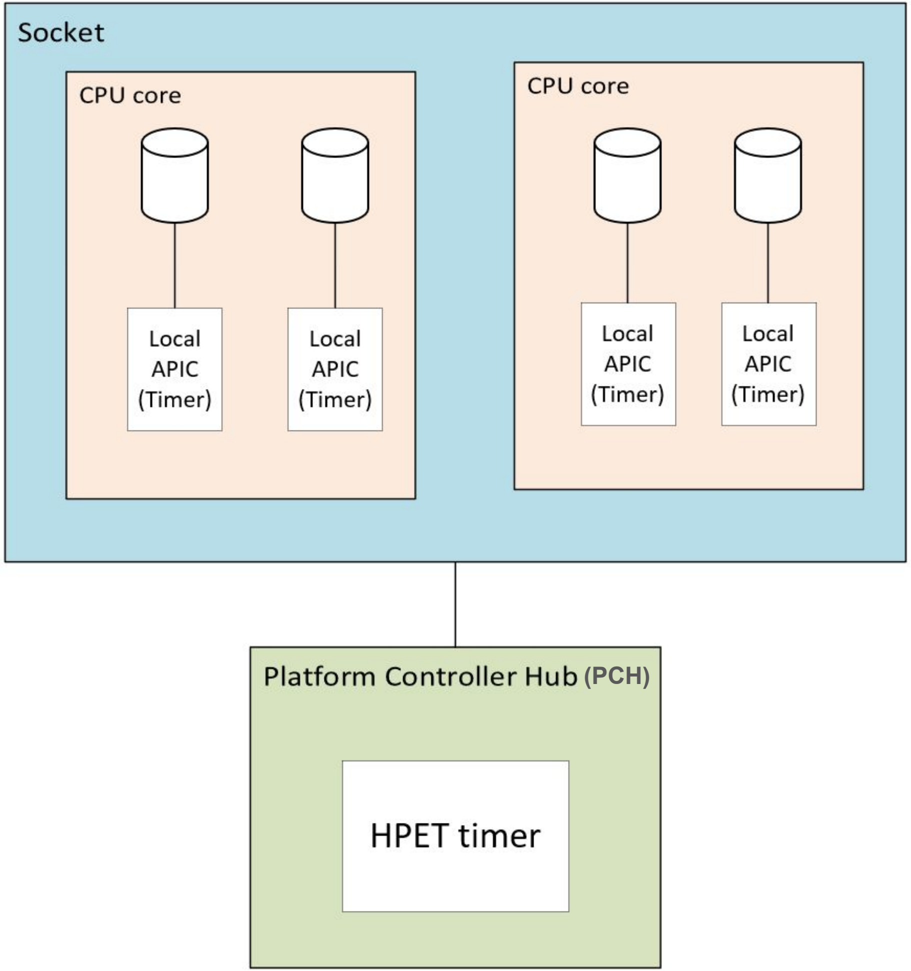
Clockevent Example: HPET

- Outside the CPU die
- Lower resolution than Local APIC (MHz).
- Applications / peripherals don’t need to depend on CPU for timing
  - Aggressive CPU power management states might turn off timers.
  - On systems without Deep C-states, Local APIC is preferred over HPET. See link.

Kernel Support - Clockevent

Clockevent Example: HPET

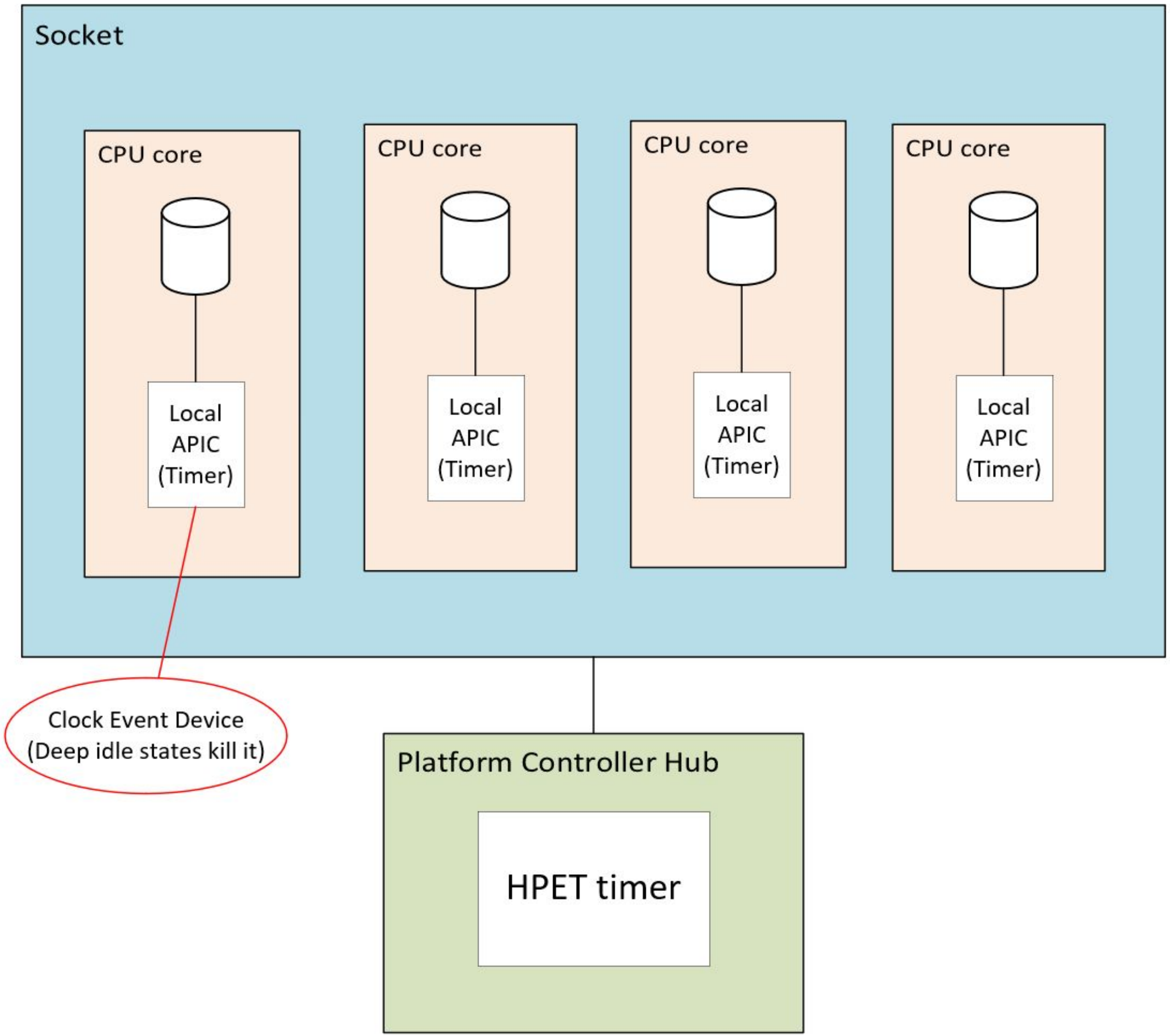
Another diagram..



Kernel Support - Clockevent

Clockevent Example: HPET

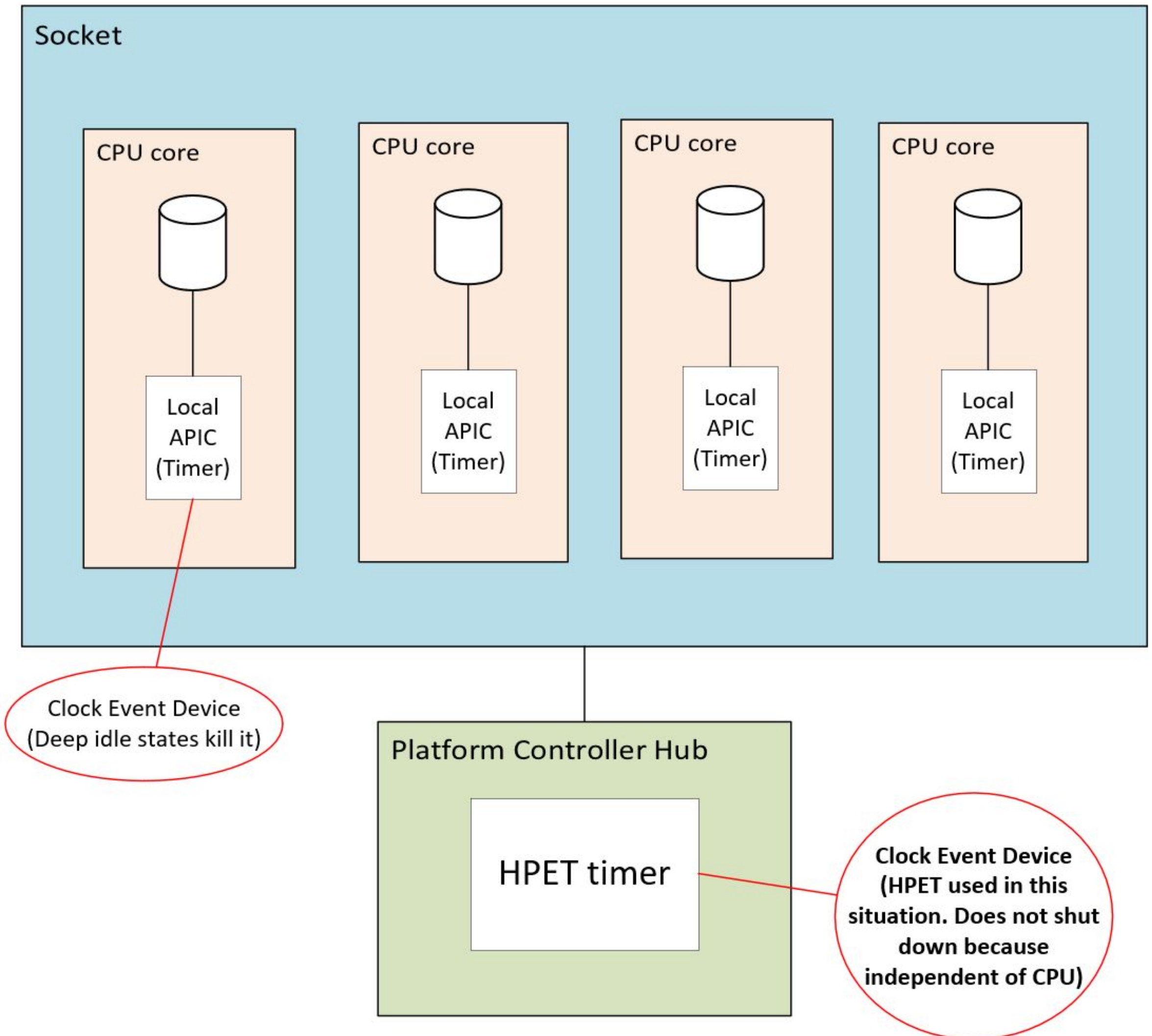
- Local APIC timer shuts down in Deeper idle states (typically C3)



Kernel Support - Clockevent

Clockevent Example: HPET

- HPET stays awake and can be used (also known as a broadcast timer)



Kernel Support - Clockevent

Clockevent Example: HPET

- This is also known as “broadcast timer”.
- To see the currently assigned broadcast timer,

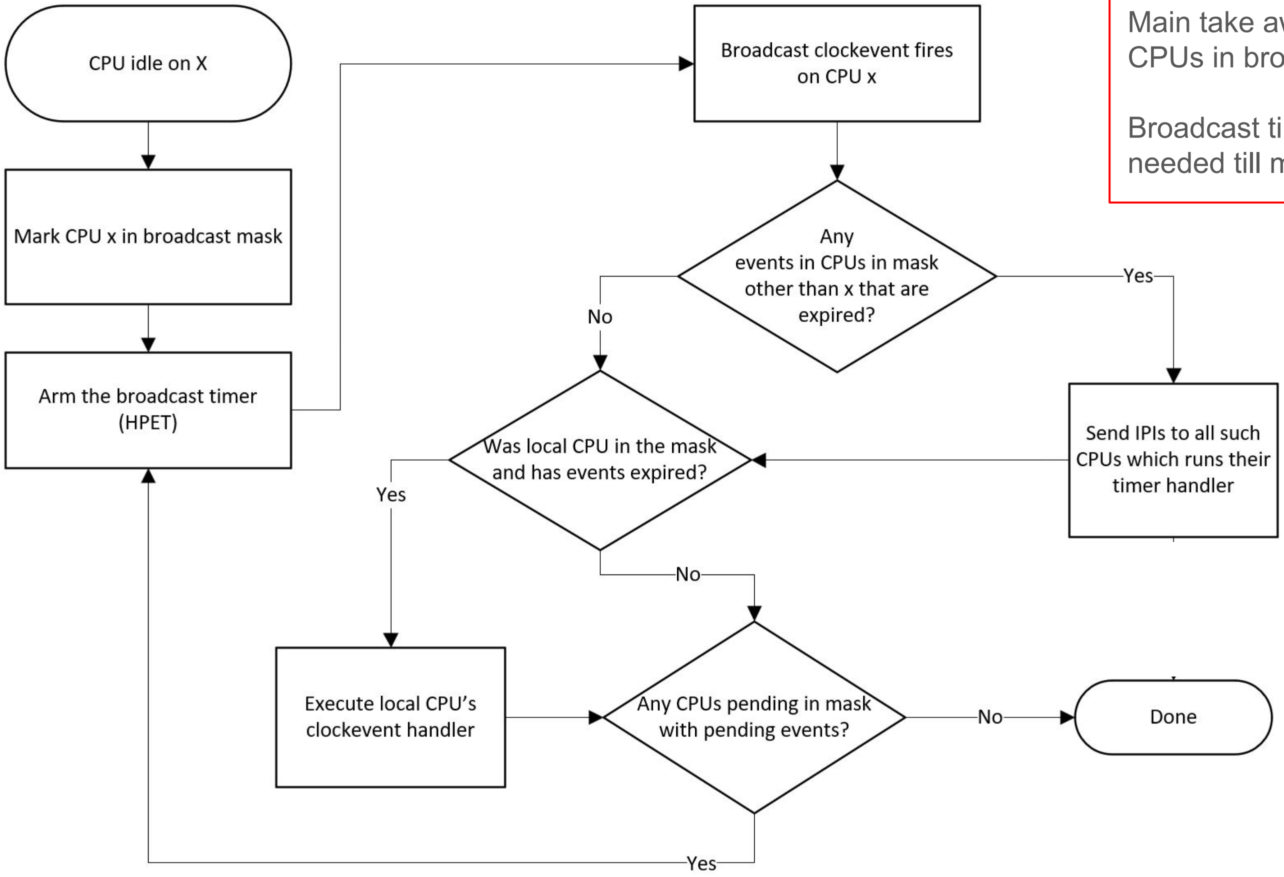
```
# cat /sys/bus/clockevents/devices/broadcast/current_device
# hpet
```

Kernel Support - Clockevent

Quiz: Obviously you have one HPET multiple CPUs that can be into deep idle state, how can that possibly work?

Just who are you kidding ???

Kernel Support - Clockevent: Broadcast Algorithm



Main take away: A CPU mask keeps track of those CPUs in broadcast mode.

Broadcast timer repeatedly fires as many times as needed till mask empty.

Kernel Support - Clockevent

More about HPET

- Can also be used as a clocksource instead of TSC.
- Can be used as a stable reference for TSC (to know if TSC is unstable).
- Slower than the TSC, not an MSR access but rather memory-mapped IO.

Kernel support - Timer wheel

Timer wheel - basic idea

- Existed from Linux early days.
- Timers that expire every 1/HZ (1 jiffy).
- Need to sort timers by order of expiry (earlier expiring timers can be queued later)
- Fast insertion, deletion expiry
    - Boils down to linked list tradeoff: Cannot have O(1) for insertion, removal and next expiry.
    - Can we gain O(1) and tradeoff space -- arrays!
- Most timer wheel users are timeouts (canceled)

Kernel support - Timer wheel

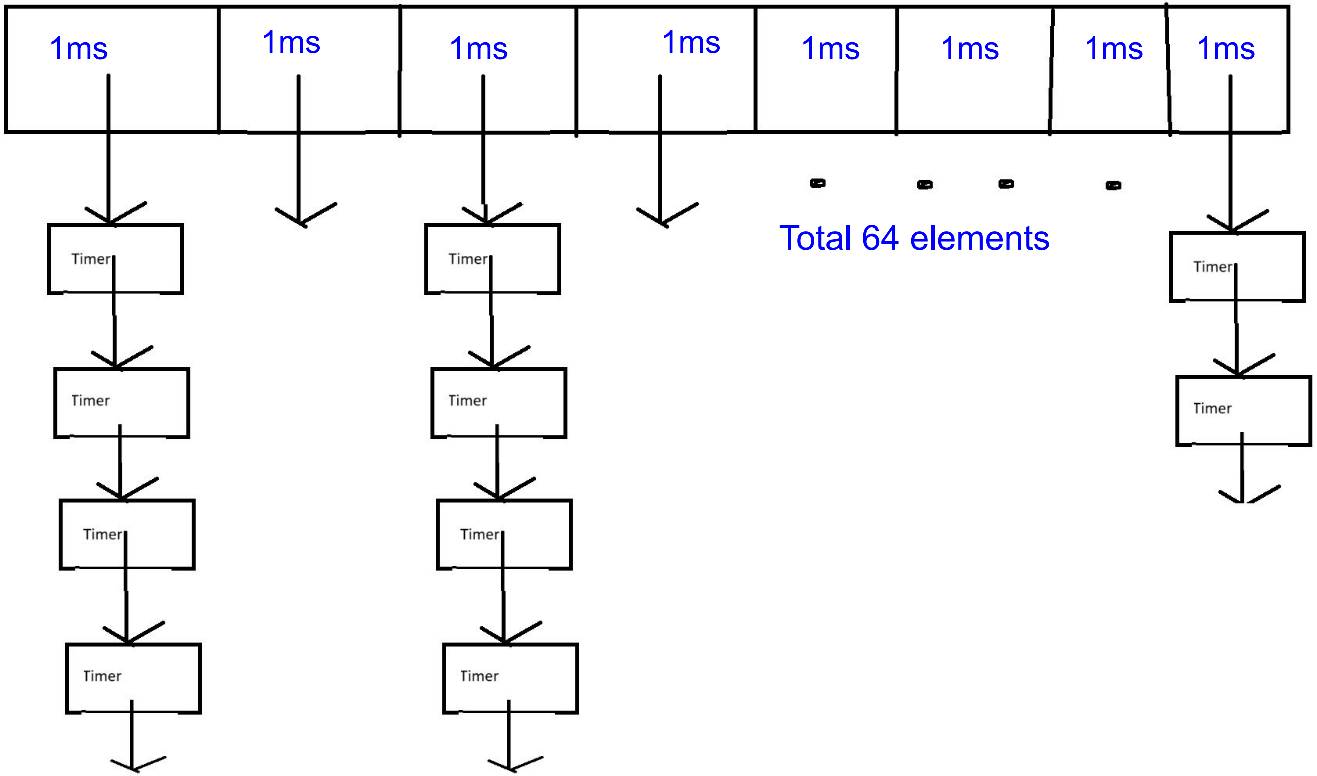
How would you design and timer subsystem?

- Need to sort timers by order of expiry (earlier expiring timers can be queued later)
- Fast insertion, deletion expiry
  - Tradeoff: Cannot have O(1) for insertion, removal and next expiry with linked list!
  - Can we gain O(1) and tradeoff space? -- arrays!
- Most timer wheel users are timeouts (canceled)

Kernel support - Timer internal implementation - timer wheel

Timer wheel FIRST level (HZ = 1000) - All timers from ~0ms to 63ms expiry are placed here

(Note the arrays are per-cpu. Timer expiry is per-cpu.)



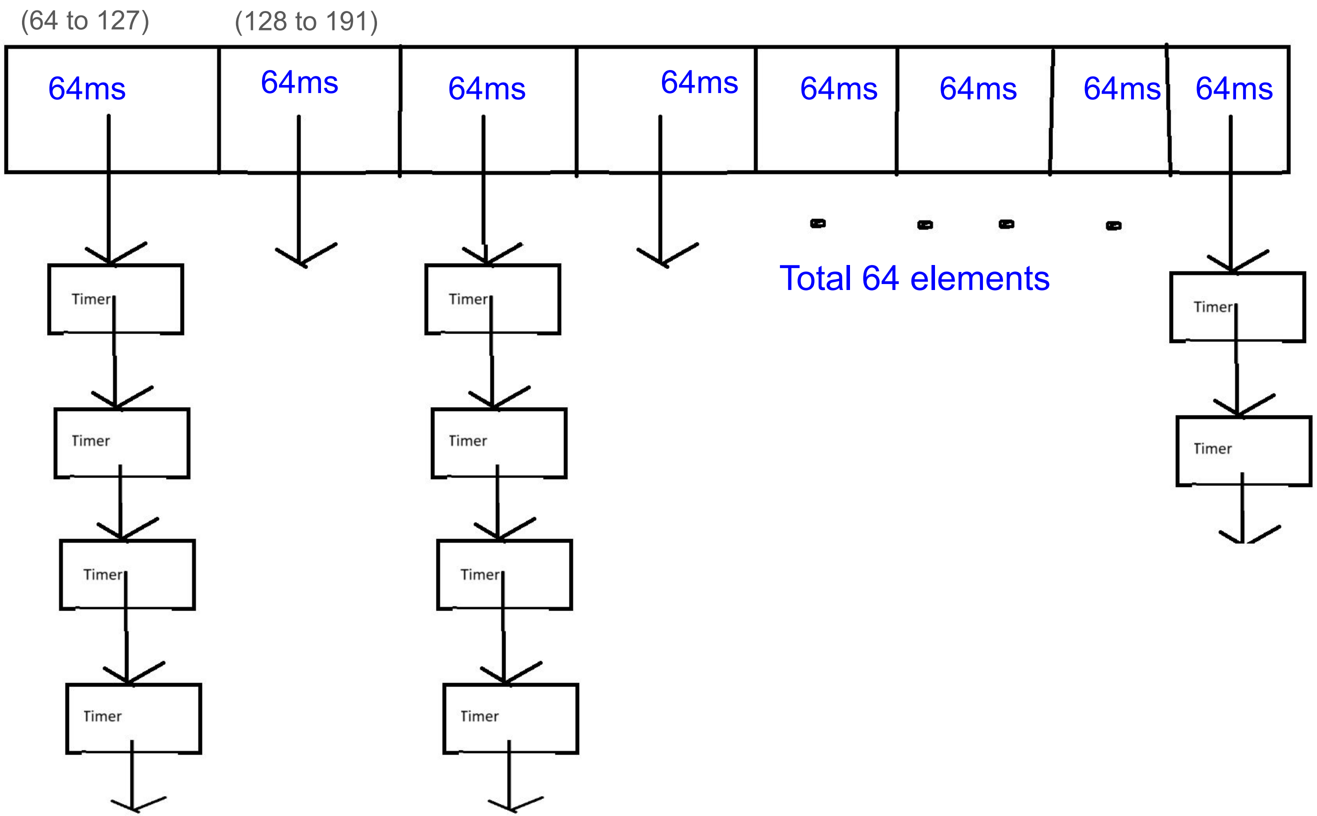
Kernel support - Timer internal implementation - timer wheel

- **Timer wheel FIRST level (HZ = 1000) - What about > 63ms, can we keep having 1ms entries?**

- **NO! Will need huge arrays!**

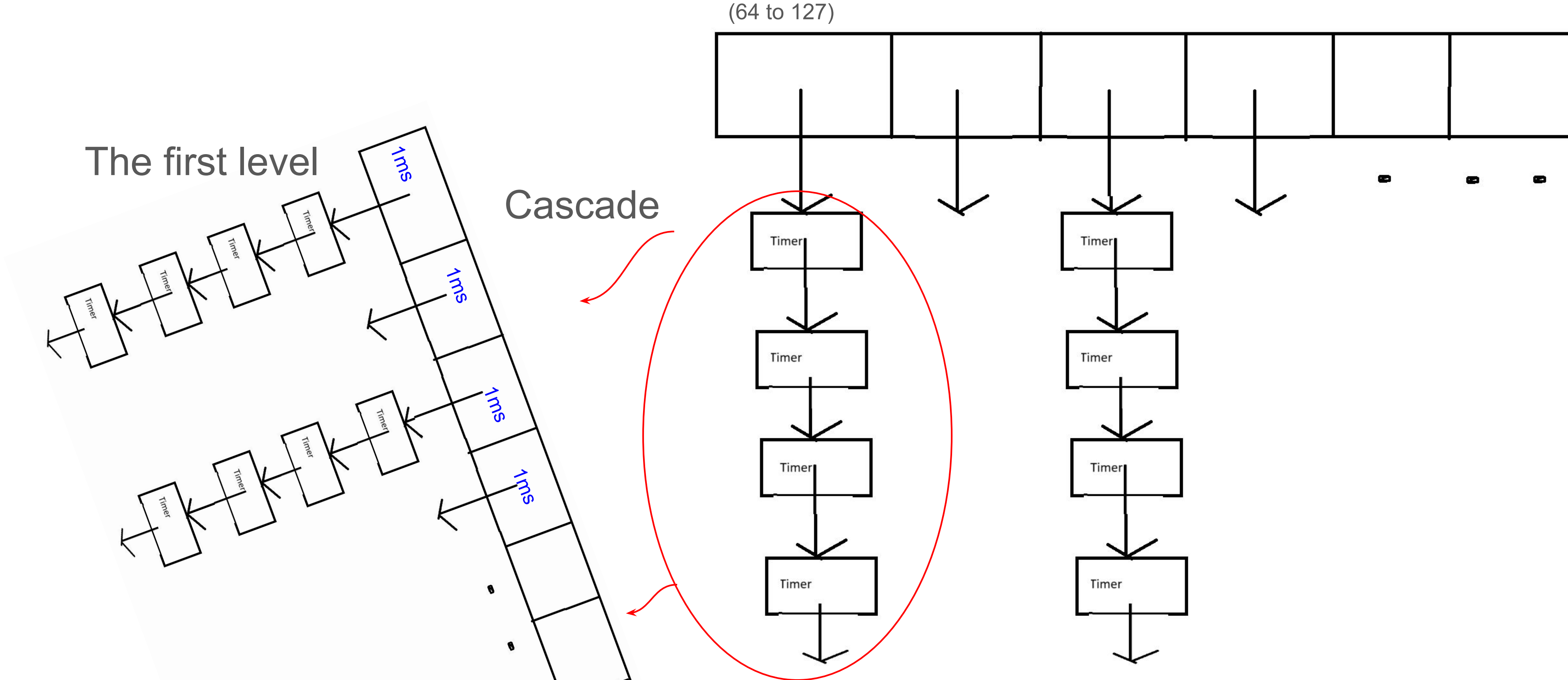
Kernel support - Timer internal implementation - timer wheel

Timer wheel SECOND level (HZ = 1000) - All timers from 64ms to 511ms expiry are placed here



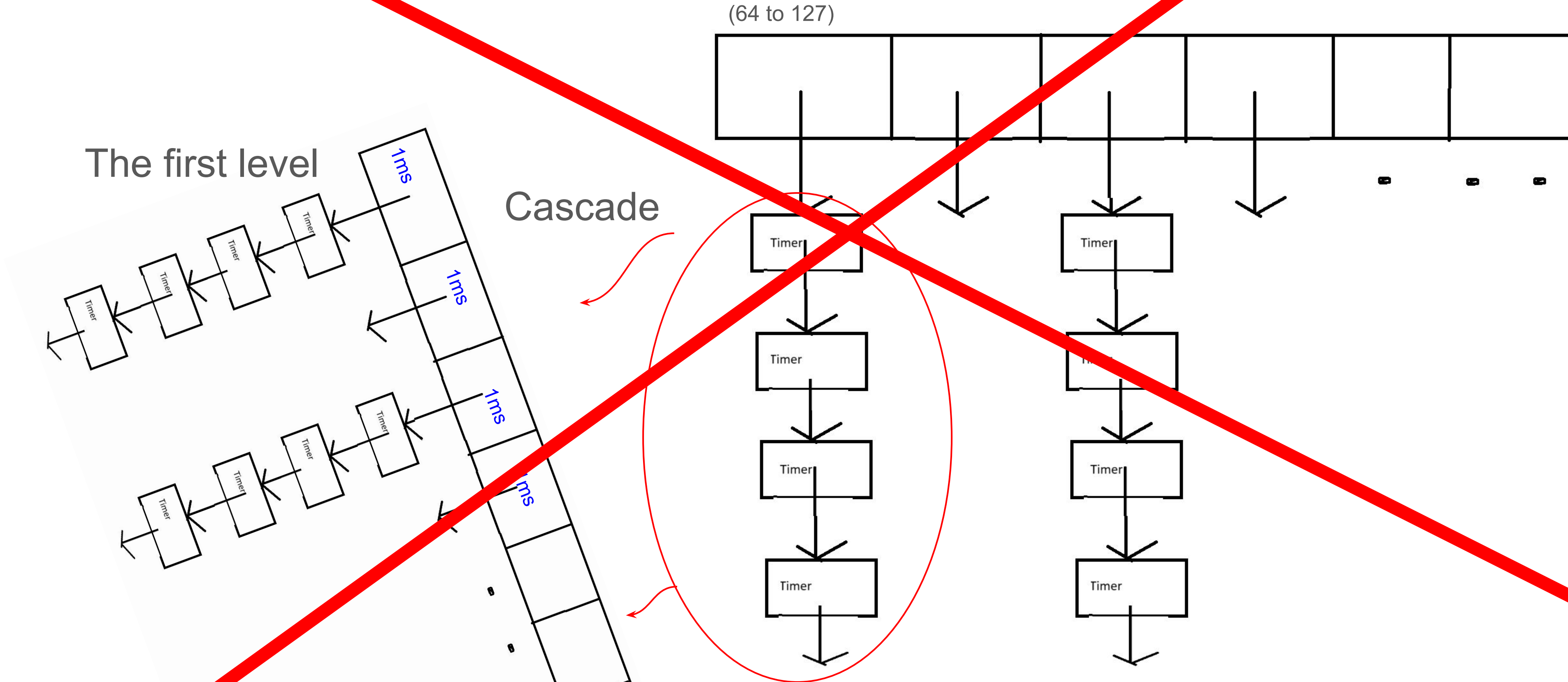
Keep moving the wheel till we hit end of first level..

Then take all timers out of first bucket of second level, move to first. Repeat.



Keep moving the wheel till we hit end of first level.

Then take all timers out of first bucket of second level, move to first. Repeat.



Cascading thought to not be worth it

- Most timers and removed before expiry, so cascading efforts wasted.
- All that while, also dirties cache lines moving timers between lists.

Kernel support - Timer internal implementation - timer wheel

No cascading of timers like before But now…

Larger the timeout, lower the granularity!

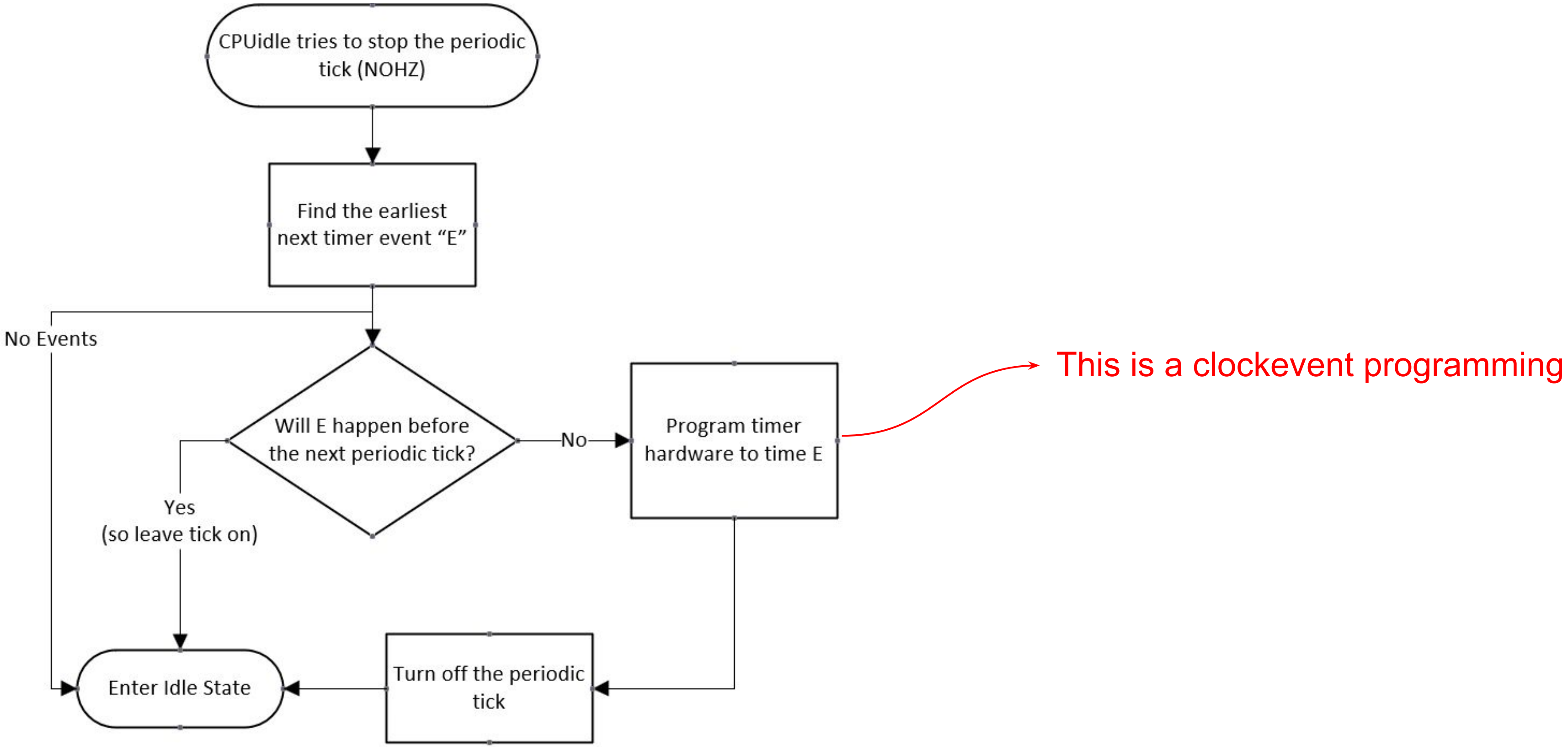
```
* HZ 1000 steps
* Level Offset  Granularity            Range
*  0      0         1 ms                0 ms -         63 ms
*  1     64         8 ms               64 ms -        511 ms
*  2    128        64 ms              512 ms -       4095 ms (512ms - ~4s)
*  3    192       512 ms             4096 ms -      32767 ms (~4s - ~32s)
*  4    256      4096 ms (~4s)      32768 ms -     262143 ms (~32s - ~4m)
*  5    320     32768 ms (~32s)    262144 ms -    2097151 ms (~4m - ~34m)
*  6    384    262144 ms (~4m)    2097152 ms -   16777215 ms (~34m - ~4h)
*  7    448   2097152 ms (~34m)  16777216 ms -  134217727 ms (~4h - ~1d)
*  8    512  16777216 ms (~4h)  134217728 ms - 1073741822 ms (~1d - ~12d)
```

Kernel support - Scheduling Clock Interrupt

- A timer interrupt that goes at a fixed rate (HZ)
- Interval of the interrupts is a “jiffie” (1 / HZ).
- One of the primary functions of the tick is for preemptive multitasking.
- The HZ rate is a balance between overhead and responsiveness.
- “jiffies” is itself a global variable that is incremented by a designated CPU.

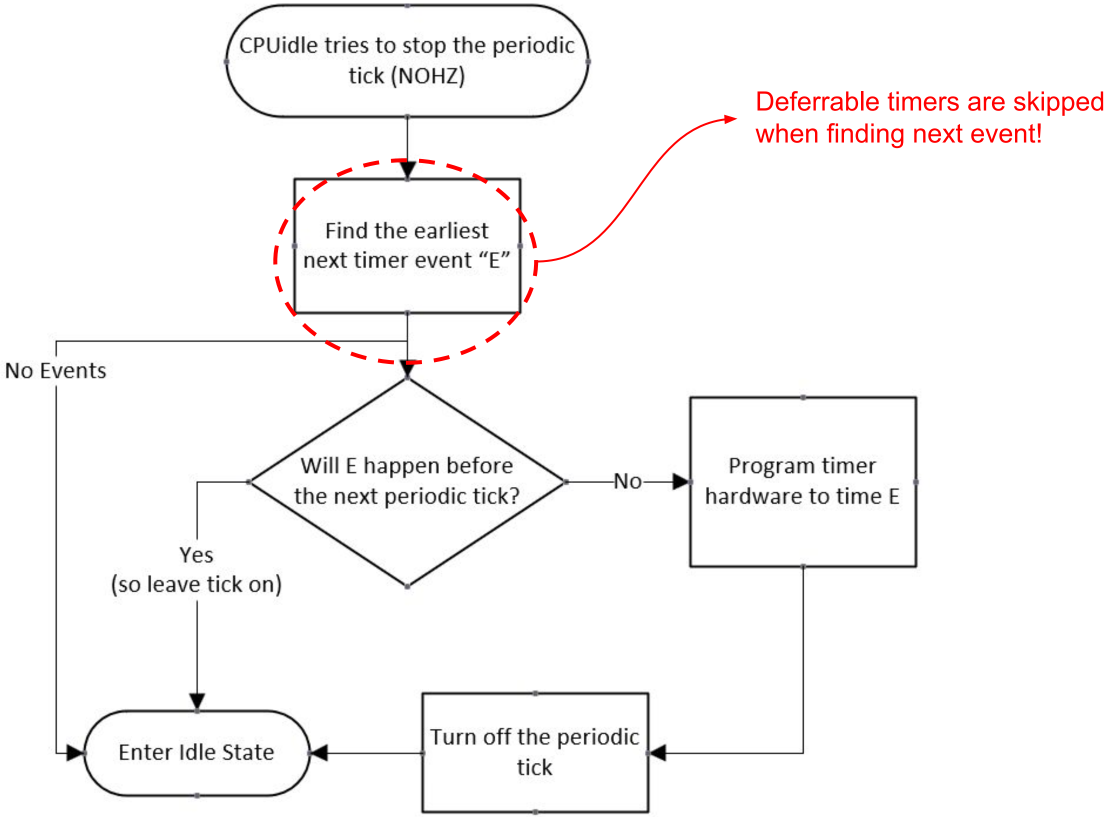
Kernel support - Deferrable timers (skip to 64 if no time)

A quick diagram on CPUidle trying to stop the periodic tick (NOHZ)



Kernel support - Deferrable timers

A quick diagram on CPUidle trying to stop the periodic tick



Kernel support - Deferrable timers

Deferrable timers have their own timer wheel:

Proof:

```
#ifdef CONFIG_NO_HZ_COMMON
# define NR_BASES 2
# define BASE_STD 0
# define BASE_DEF 1
#else
# define NR_BASES 1
# define BASE_STD 0
# define BASE_DEF 0
#endif


static DEFINE_PER_CPU(struct timer_base, timer_bases[NR_BASES]);
```

Kernel support - Deferrable timers

Deferrable timers initialization and firing

- Deferred timers are initialized by a call to `timer_setup()` with the `TIMER_DEFERRABLE` flag.

- The per-cpu clock event which is programmed for NON DEFERRABLE timer event fires:

```
tick_sched_handle() ->
      update_process_times() ->
            run_local_timers()
```

Kernel support - Deferrable timers

Deferrable timers initialization and firing

- When this clock event fires, it also scoops up the expired deferrable timers:

```
/*
 * Called by the local, per-CPU timer interrupt on SMP.
 */
static void run_local_timers(void)
{
	struct timer_base *base = this_cpu_ptr(&timer_bases[BASE_STD]);

	/* Raise the softirq only if required. */
	if (time_before(jiffies, base->next_expiry)) {
		if (!IS_ENABLED(CONFIG_NO_HZ_COMMON))
			return;
		/* CPU is awake, so check the deferrable base. */
		base++;
		if (time_before(jiffies, base->next_expiry))
			return;
	}
	raise_softirq(TIMER_SOFTIRQ);
}
```

Kernel support - Deferrable timers

Deferrable timers initialization and firing

- Finally, the timer softirq runs the deferred timers as well.

```
static void run_timer_softirq(struct softirq_action *h)
{
	struct timer_base *base = this_cpu_ptr(&timer_bases[BASE_STD]);

	__run_timers(base);
	if (IS_ENABLED(CONFIG_NO_HZ_COMMON))
		__run_timers(this_cpu_ptr(&timer_bases[BASE_DEF]));
}
```

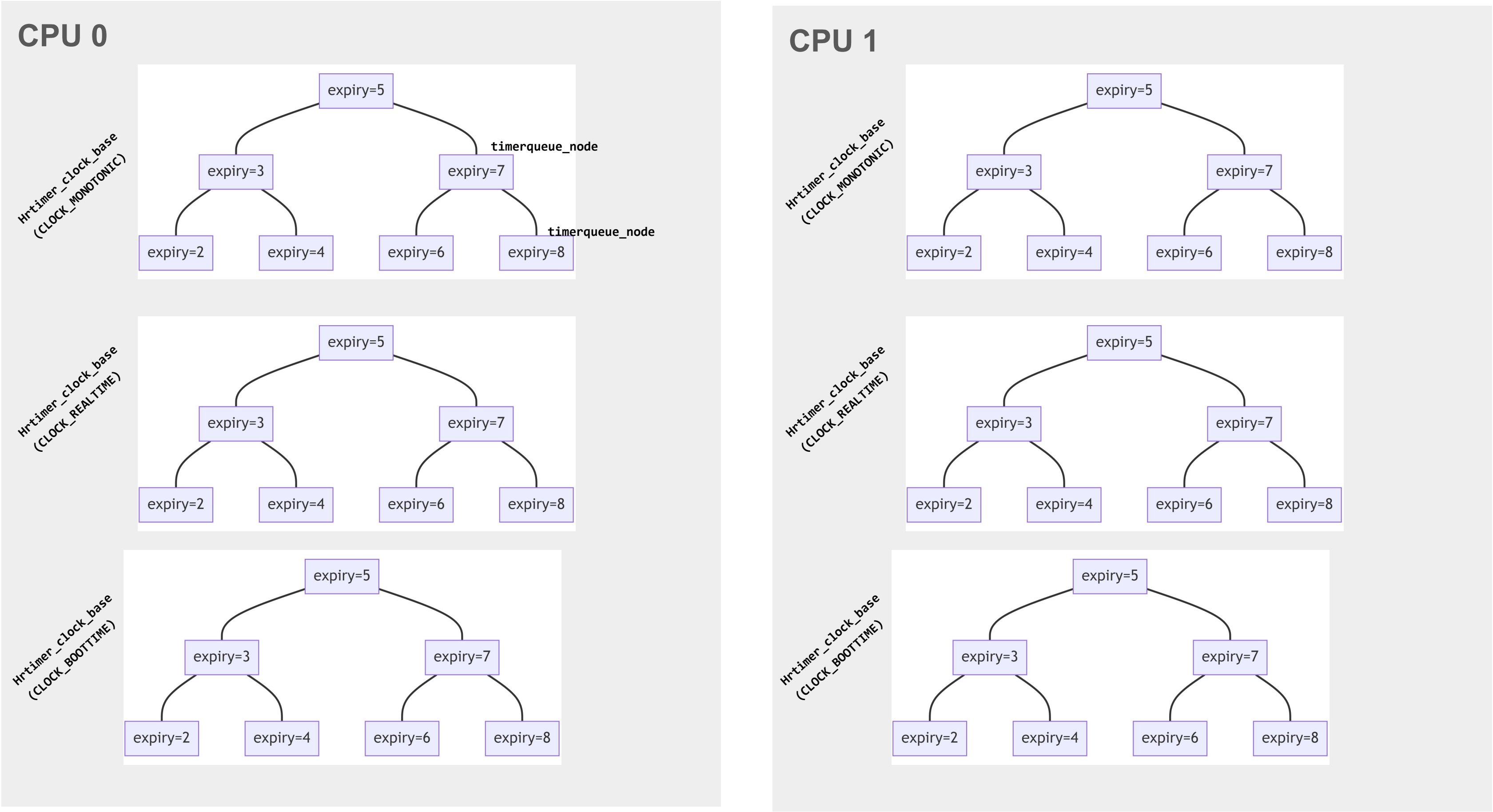
Kernel support - Deferrable timers

Example of a Deferrable timer user:

```
static int init_worker_pool(struct worker_pool *pool)
{
   [...]
   timer_setup(&pool->idle_timer, idle_worker_timeout, TIMER_DEFERRABLE);
   [...]
}

/**
 * idle_worker_timeout - check if some idle workers can now be deleted.
 * @t: The pool's idle_timer that just expired
 */
static void idle_worker_timeout(struct timer_list *t)
```

Kernel support - high resolution timers (hrtimer)

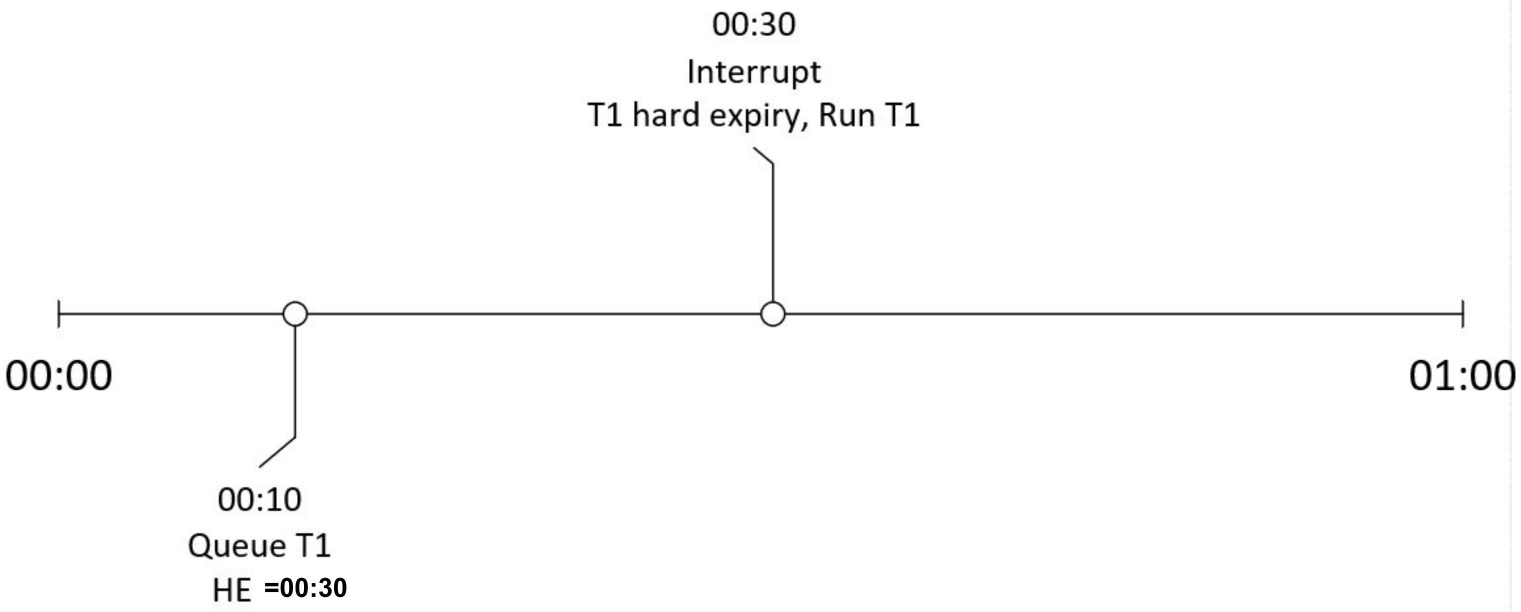


Kernel support - high resolution timers (hrtimer)

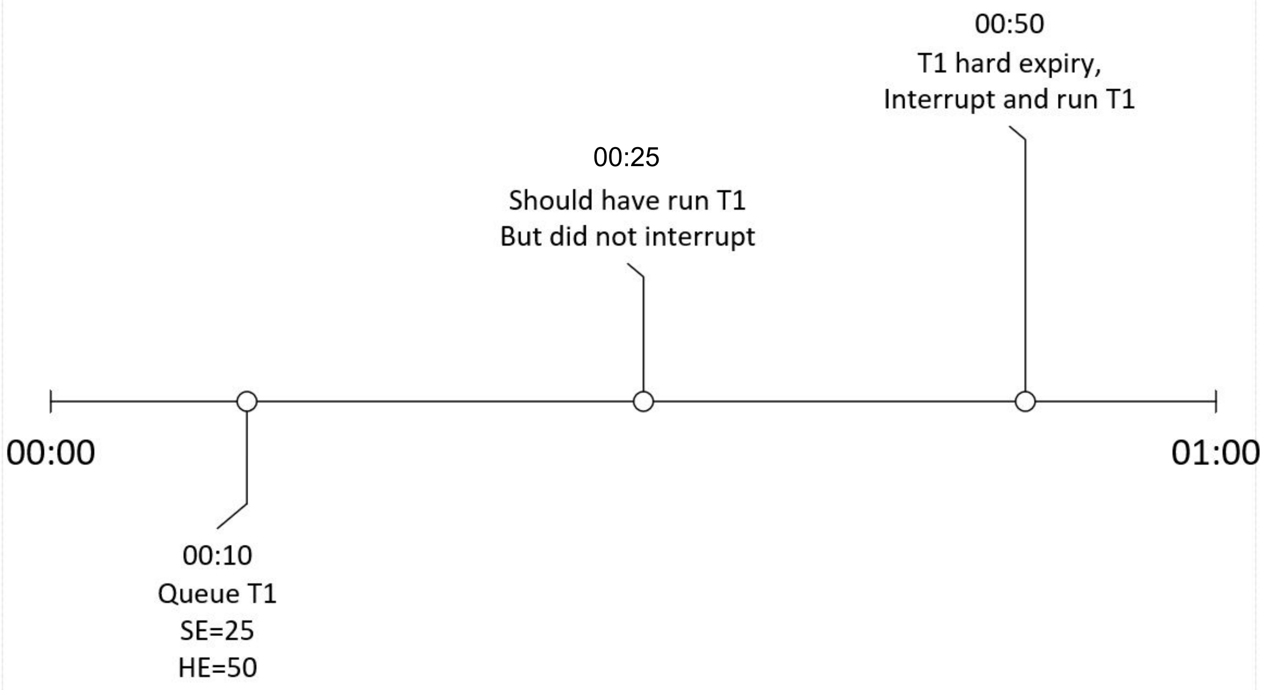
Hrtimer Range timers :   Hrtimers can be queued with some “timer slack”

- Normal hrtimers will have both a soft expiry and hard expiry **which are equal** to each other.
- But hrtimers with slack will have a soft expiry & hard expiry which is the soft expiry + delta.
- The idea is to reduce wakeups and save power.

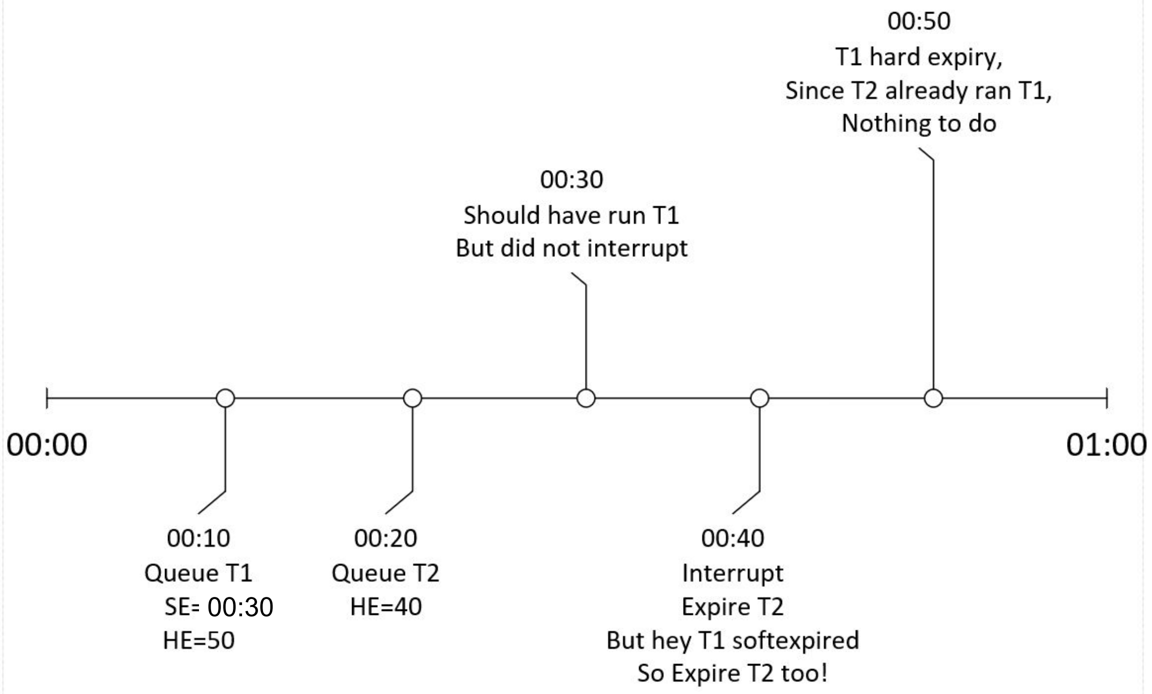
Normal HRtimer without slack



HRtimer with slack expires after soft, AT hard expiry..

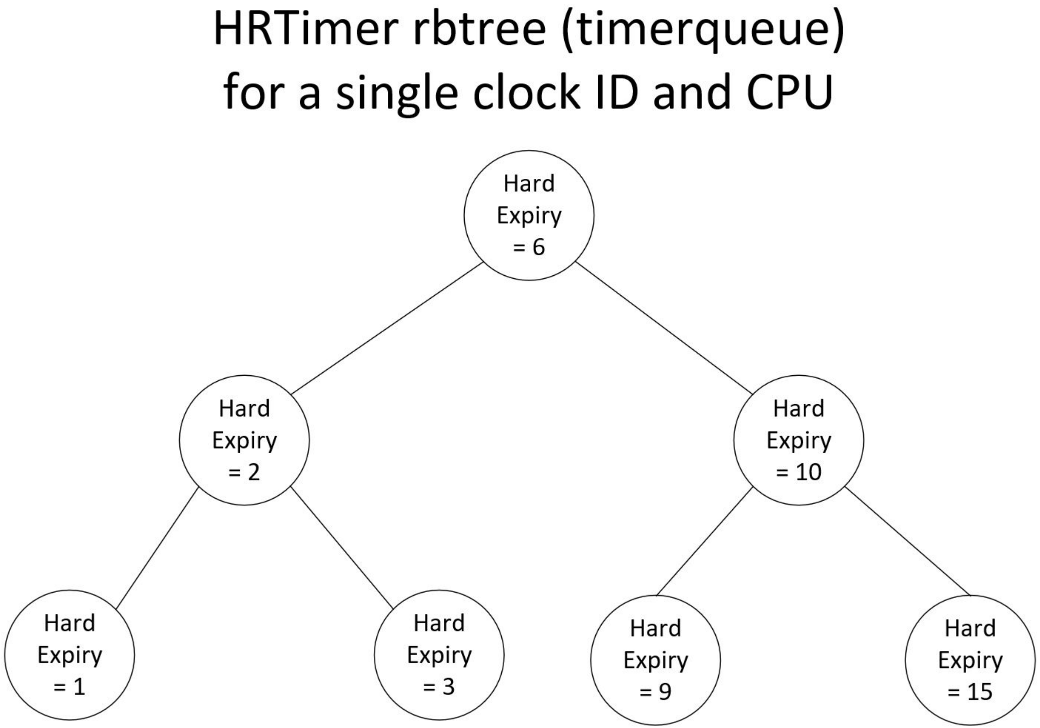


HRtimer with slack expires after soft, before hard expiry.



Kernel support - high resolution timers (hrtimer)

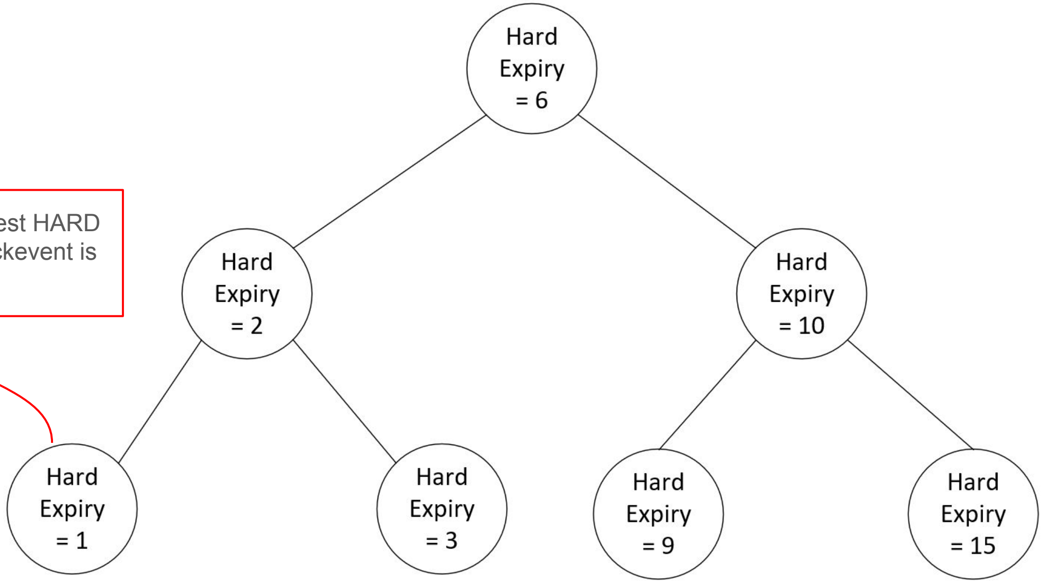
Diagram of a single rbtree.



Kernel support - high resolution timers (hrtimer)

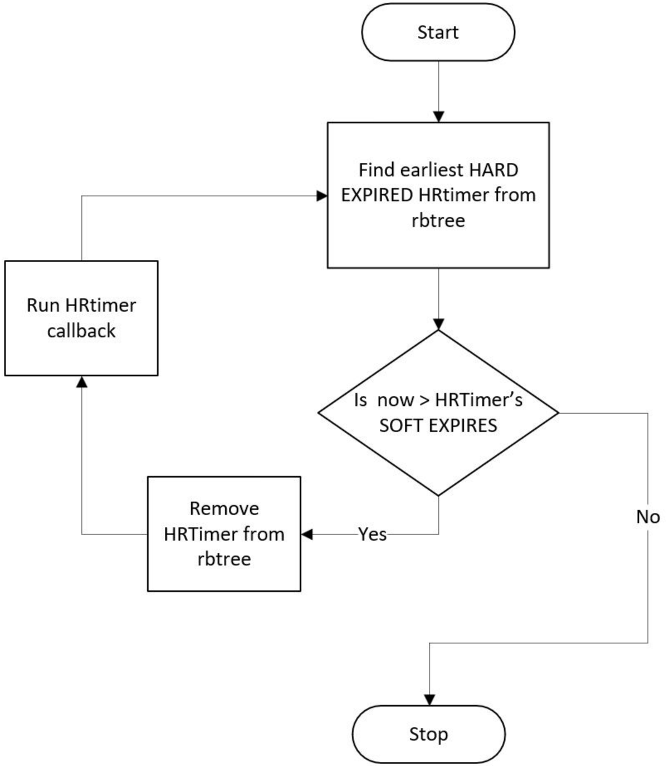
Diagram of a single rbtree.

HRTimer rbtree (timerqueue)
for a single clock ID and CPU



Note: The earliest HARD expiry time clockevent is programmed!

Simplified algorithm for HRTimer expiry



Remember that for normal (non-slack) timers, hard exp == soft exp time.

A soft expired timer may not always execute when a hard expired one runs.

- `Consider the situation of hard expiries in the timerqueue: [ 5, 10, 20, 30, 40 ]`
- `The corresponding soft expiries for these are            : [ 5, 10, 9,  30,  8 ]`
- `Notice that the 3rd and 5th timers are slack (hard expiry != soft expiry)`

Say the second timer (a non-slack one) is currently expiring and the time is now T=10.

Now, since the 3rd timer’s soft expiry is 9, that is expired as well.

BUT, timer 5 has also expired and is not considered because we break out of the loop due to timer 4. So, In theory that could have been run but its not!

Kernel support - high resolution timers (hrtimer)

Main takeaways:

- Nanosecond resolution instead of jiffies (but depends on hardware, IRQ delays etc)
- Higher overhead for insertion, removal than wheel.
- Required for Real Time workloads which need high resolution (cyclictest is a test).
- Different POSIX clocks have their own rbtree.
- Further, all the rbtrees are duplicated for each CPU.
- Timer slack can save power by reducing number of interruptions and coalescing.
- Soft expired timers may not always run even if they could.
- HRtimers are not deferrable unlike timerwheel ones.

Users of Timer wheel vs HRtimers

Use case	Infra
schedule_timeout(jiffies) -- synchronous sleep jiffies until timeout (used a lot in net, fs, gfx, RCU etc.)	timer wheel
Networking, filesystems, misc timeouts	timer wheel
RCU internal machinery	timer wheel
Futex timeout (syscalls accept clockids)	hrtimer
Nanosleep syscall	hrtimer
POSIX clocks, timers, timerfd APIs	hrtimer
Scheduler tick in high res mode	hrtimer (sched_timer)
Timer wheel expiries in high res mode	hrtimer (sched_timer)

Comparison of Timer wheel vs HRtimers

	Timer wheel	**HRtimer**
Resolution	Jiffy (1/HZ)	Nanoseconds.
Insert/Deletion Overhead	O(1)	O(log)
Number of IRQs	Low	High
Can be turned off	No	Yes (low res mode, HRtimer API still effective at low accuracy).

Back to the periodic tick..

- Now let us get into the periodic tick.
- Also known as the tick.
- Also known as the scheduling clock interrupt.

Kernel support - Scheduling Clock Interrupt

- A timer interrupt that goes at a fixed rate (HZ)
- Interval of the interrupts is a “jiffie” (1 / HZ).
- One of the primary functions of the tick is for preemptive multitasking.
- The HZ rate is a balance between overhead and responsiveness.
- “jiffies” is itself a global variable that is incremented by a designated CPU every 1/HZ.

Recall… Kernel Support - Clockevents

A clockevent device abstracts a device which generates interrupt at programmed time in the future.

```
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);
```

Run callback on next event.

Clock event features. ONESHOT is required for NOHZ

Which handler is run depends on the "tick mode" of the system.

The tick internals

Different handler for different tick modes.

Handler	**Usage**
`tick_handle_periodic()`	Periodic mode
`tick_nohz_handler()`	Low res mode
`hrtimer_interrupt()`	High res mode

The tick internals

Periodic mode

`tick_handle_periodic()` - Ticks **even during Idle**

```
|-----|-----|-----|------|-----|-----|-----|-----|-----|-----|------> time
   T1   idle   idle   idle    t2     t3
```

The tick

`Key:`
`t1, t2, t3:` tick's timer expiry periods.
`idle:` Idle periods

The tick internals

Low resolution mode (Tickless and low res)

`tick_nohz_handler()` - No ticks during idle

```
|-----|-----------------|-----|-----|-----|-----|-----|-----|----> Time
   t1          idle        t2     t3

      The tick
```

```
Key:
t1, t2, t3: tick’s timer expiry periods.
idle: Idle periods
```

- Also known as NOHZ mode (CONFIG_NOHZ_IDLE).
- Requires one-shot mode in clockevent! (fire once in the future at dynamic point)

The tick internals

Note!

In periodic and low res mode: The scheduling clock interrupt handles both Timer wheel and HR timer events!

The tick internals

Comparison between periodic and low res mode

Mode	Periodic	Low Res
Minimum timer resolution	1 / HZ	1 / HZ

What if need lower resolution

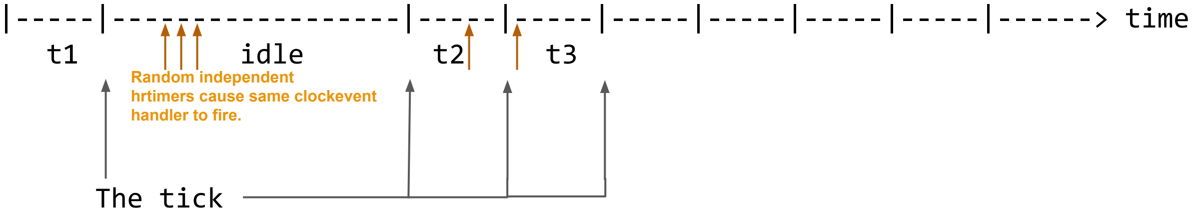
We already have ONESHOT capable clockevent for low res.

We can program that to fire at any time (nanosecond) in future, not just at 1/HZ.

The tick internals

High resolution mode (Tickless and high res)

`hrtimer_interrupt()` - No ticks during idle + independent highres irqs.



(In highres mode, The tick is itself an hrtimer!)

```
Key:
t1, t2, t3: tick's timer expiry periods.
idle: Idle periods
```

- Compatible with NOHZ mode (CONFIG_NOHZ_IDLE).
- Needs a clockevent device capable of firing in one-shot mode (fire once in the future at dynamic point)

The tick internals

Comparison between periodic, low and high res mode

Mode	Periodic	Low Res	High Res
Ticks during idle	Yes	No	No
Minimum timer resolution	1 / HZ	1 / HZ	Nanosecond
Hrtimer API	Still works but low in res.	Still works but low in res.	High resolution
Power Savings	Bad	Good	Ok
Practical	No	Could be	Yes
Requires ONE SHOT clockevent	No	Yes	Yes

Kernel support - NOHZ - Turn off the tick

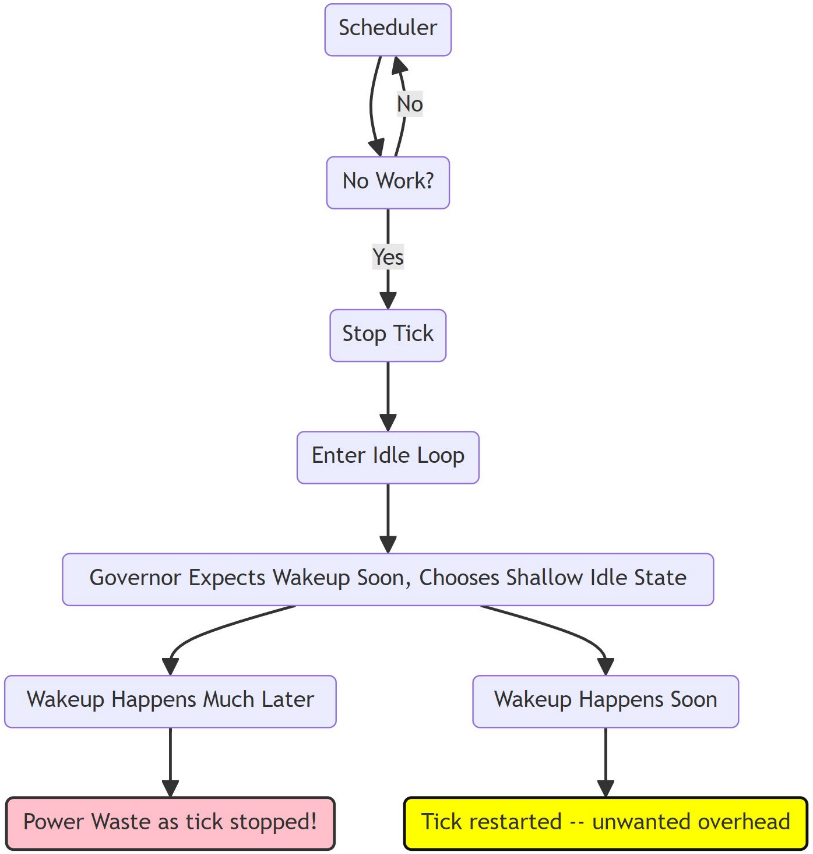
- Tick does not need to run when CPU is idle, it wastes power.
- CPUidle governor makes a decision about turning off the tick.
- `CONFIG_NO_HZ_IDLE` turns off tick when CPU is idle.
- `CONFIG_NO_HZ_FULL` turns off tick if only 1 task is active or CPU idle.

Kernel Support - CPUidle governor and Tick Stop

Old kernels:

Stop tick,
Then choose
Idle state

(Governor
Doesn't
Stop the tick)



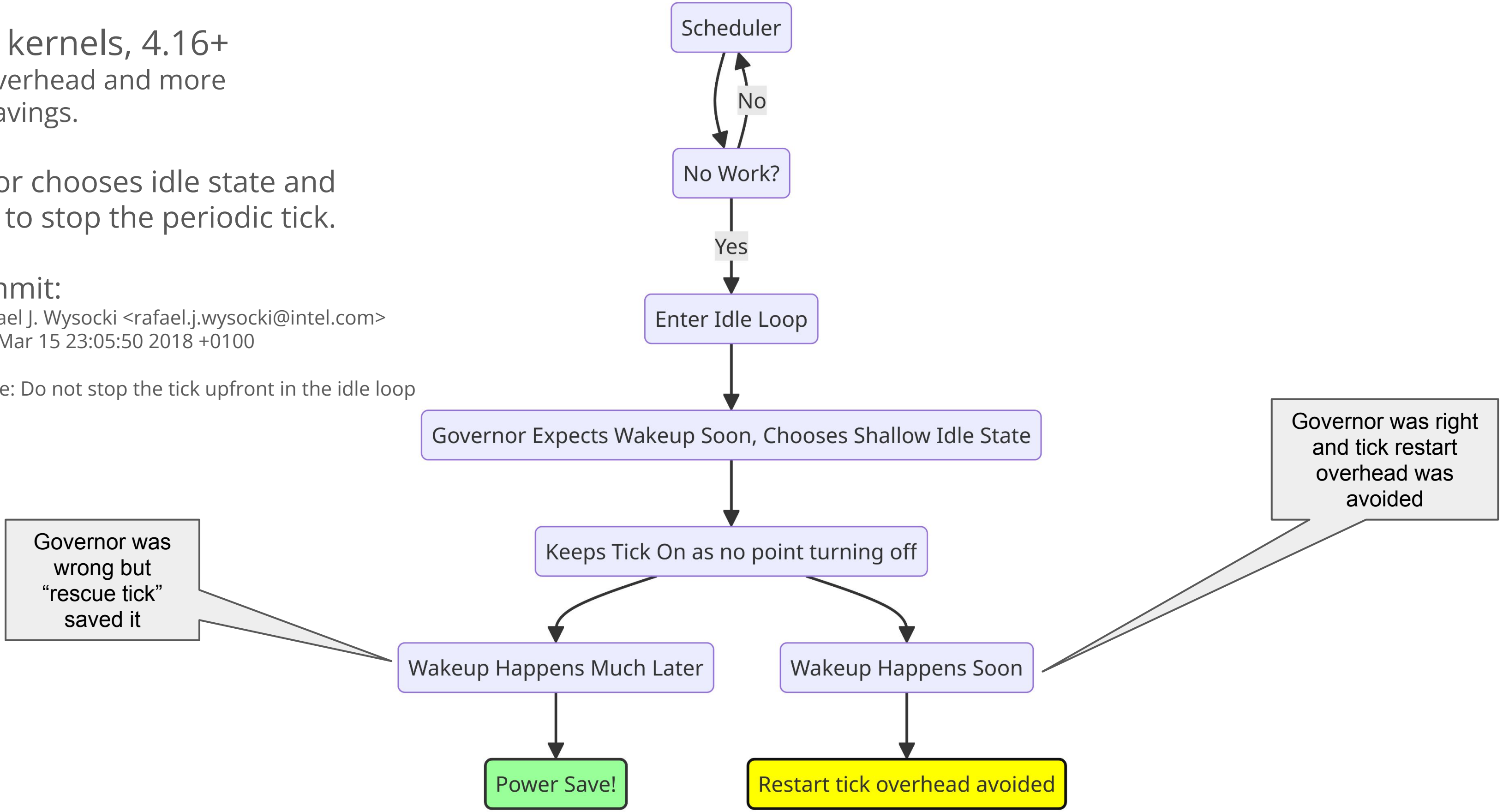
Kernel Support - CPUidle governor and Tick Stop

Newer kernels, 4.16+
Lower overhead and more
Power savings.

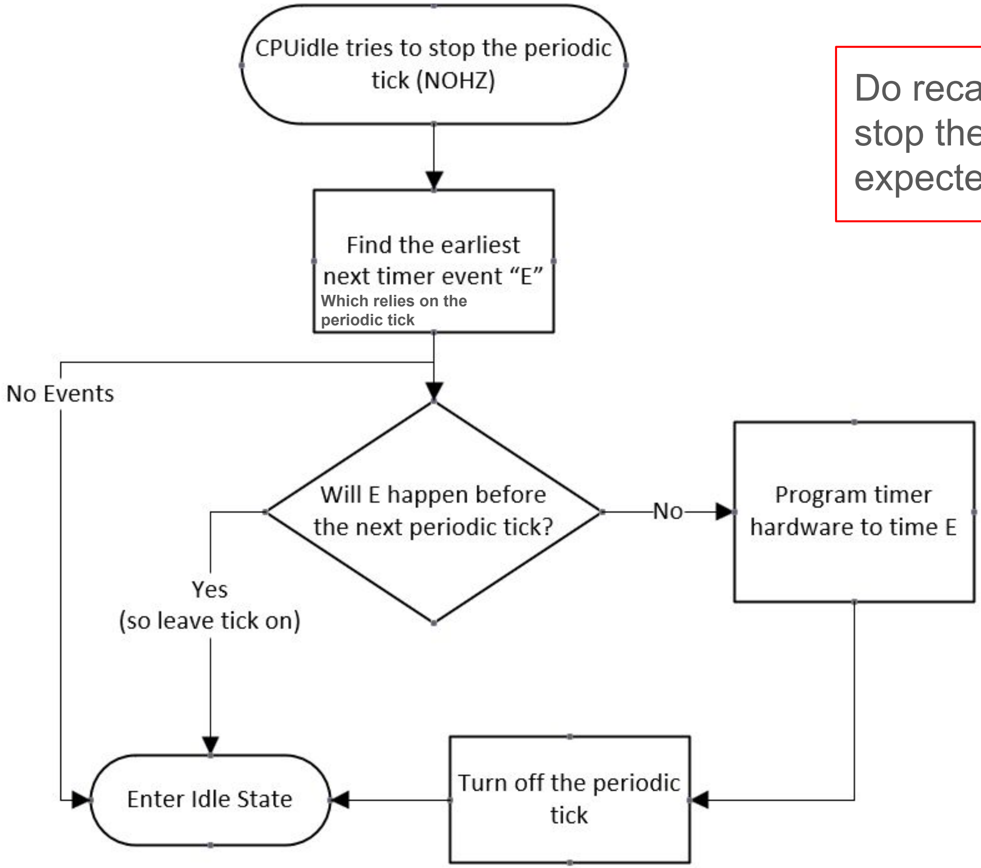
Governor chooses idle state and
decides to stop the periodic tick.

See commit:
Author: Rafael J. Wysocki <rafael.j.wysocki@intel.com>
Date: Thu Mar 15 23:05:50 2018 +0100

sched: idle: Do not stop the tick upfront in the idle loop



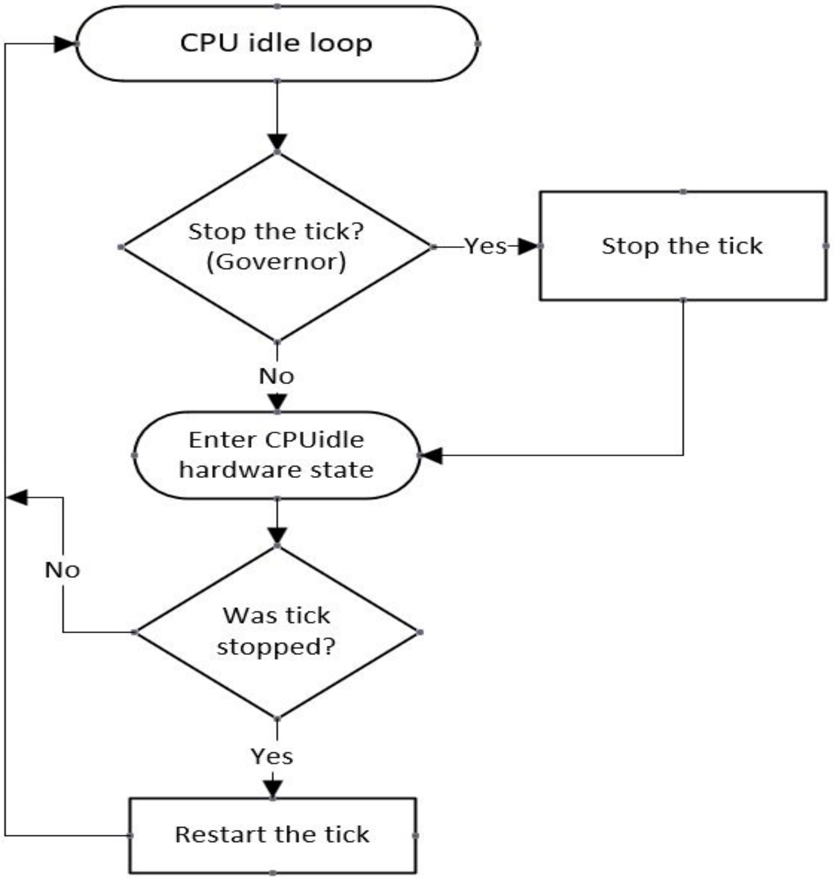
NOHZ code is boss! Final decision for tick shutdown left to it. (Governor just hints)



Do recall that even though governor decided to stop the tick, if there is an imminent timer event expected,the periodic tick will be left on…

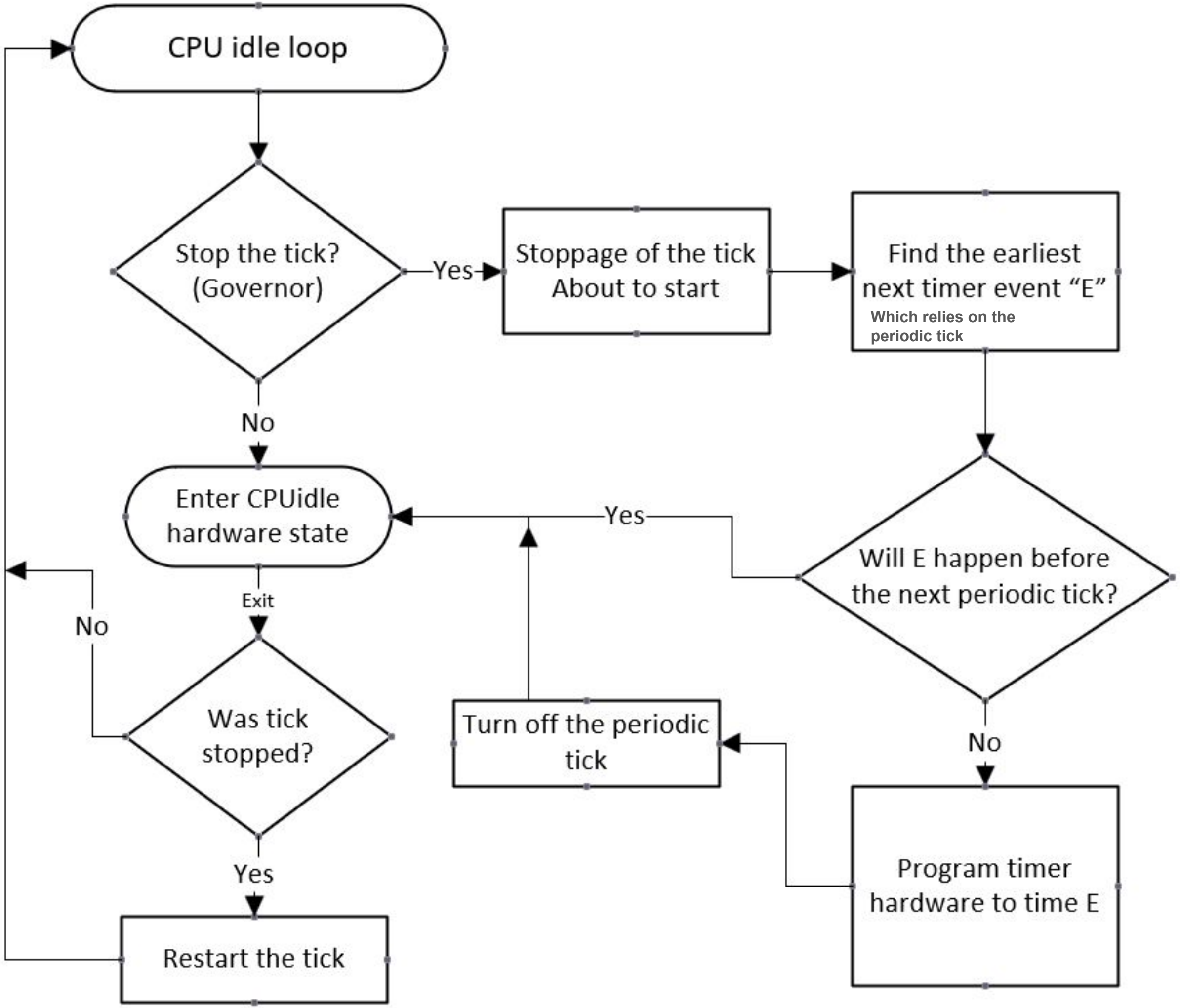
Kernel support - NOHZ - Periodic Tick restart

Tick is unconditionally restarted upon exiting from CPU idle state



Putting it Together…

The periodic tick lifecycle.



VDSO

- Some timekeeping syscalls are available as VDSO, like `clock_gettime()`. Huge perf benefit.
- Kernel maps code and data into user space to allow direct calls, not supported on all architectures.
- VDSO mapping is an ELF object, similar to a dynamic library, mapped into user space with a dynamic symbol table for function location.

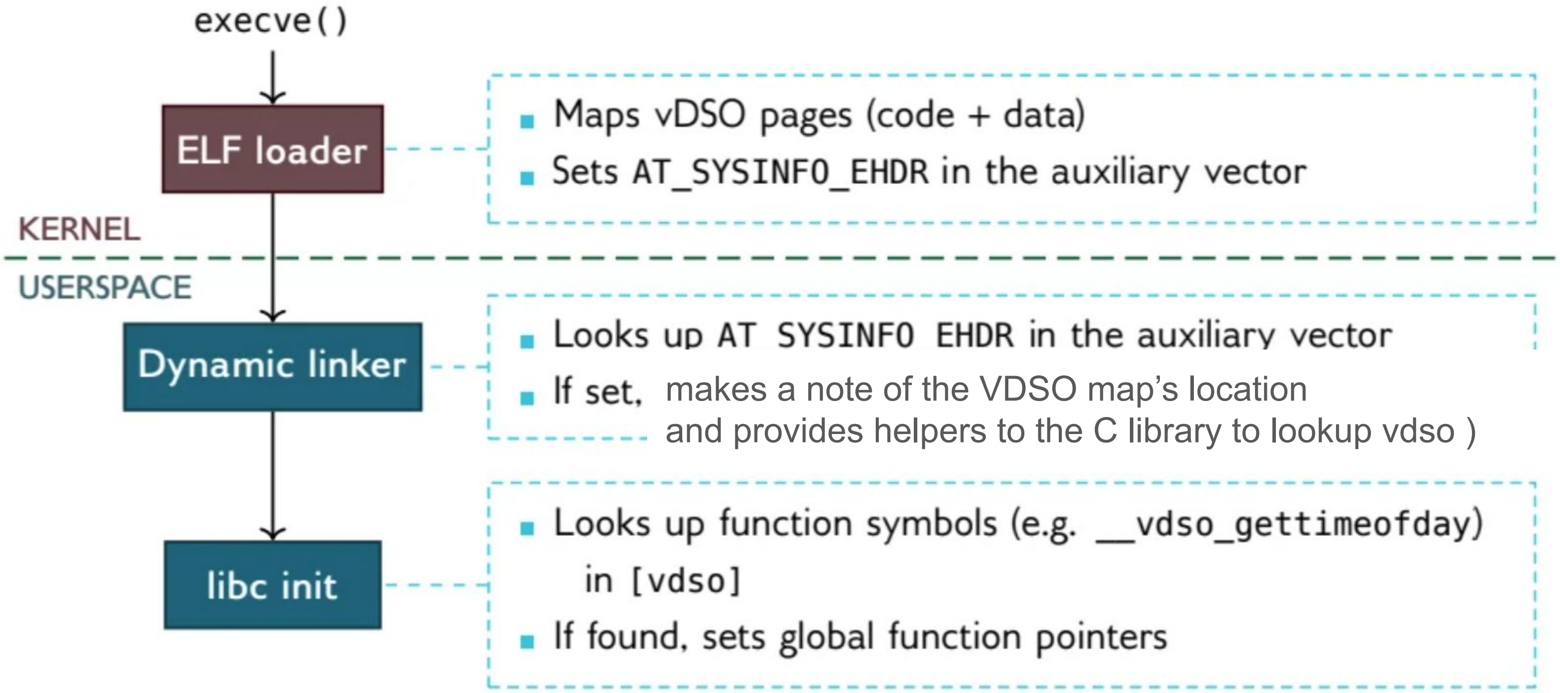
Example, for `clock_gettime() VDSO implementation`:

- VDSO mapping contains a **`struct vdso_data`** in the data page for time calculation, including base time, last TSC value, and slope.
- Users calculate current time by reading the TSC and applying the formula:

  `Current time = base time + (current TSC - last cycle) * slope.`

VDSO

Kernel and userspace setup



Thank you! It is time!

Until another time… ;-)   -Joel