UNIX Clocks (x86) - Top-Down View

```
TLDR: calling "clock_gettime" with a clockid_t in
    CLOCK_RUNTIME
    CLOCK_MONOTONIC
    CLOCK_MONOTONIC_RAW
    CLOCK_PROCESS_CPUTIME_ID
will very likely be based on the TSC on x86 architectures. Although not explored
here explicitly, CLOCK_BOOTTIME and CLOCK_TIME indirectly rely on the same clock:
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/timekeeping.c#L708
The same holds for CLOCK_THREAD_CPUTIME_ID.
```

Trapping Into The Kernel

```
The <time.h> "int clock_gettime(clockid_t, struct timespec *)":
http://git.musl-libc.org/cgit/musl/tree/src/time/clock_gettime.c
takes in a "clockid_t" (e.g. CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_MONOTONIC_RAW,
CLOCK_PROCESS_CPUTIME_ID) and reads the current time (in seconds and nanoseconds)
into the "struct timespec" passed as an argument.
```

```
int __clock_gettime(clockid_t clk, struct timespec *ts)
{
    int r;
    ...
#ifdef SYS_clock_gettime64
    r = -ENOSYS;
    if (sizeof(time_t) > 4)
        r = __syscall(SYS_clock_gettime64, clk, ts);
    if (SYS_clock_gettime == SYS_clock_gettime64 || r!=-ENOSYS)
        return __syscall_ret(r);
    long ts32[2];
    r = __syscall(SYS_clock_gettime, clk, ts32);
    ...
#endif
    if (!r) {
        ts->tv_sec = ts32[0];
        ts->tv_nsec = ts32[1];
        return r;
    }
    return __syscall_ret(r);
    ...
}
```

```
The implementation is a thin wrapper around the "SYS_clock_gettime" system call:
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/posix-timers.c#L1027
```

```
SYSCALL_DEFINE2(clock_gettime, constclockid_t, which_clock,
        structtimespec__user *,tp)
{
    structk_clock *kc = clockid_to_kclock(which_clock);
    structtimespec kernel_tp;
    int error;

    if (!kc)
        return -EINVAL;

    error = kc->clock_get(which_clock, &kernel_tp);

    if (!error &&copy_to_user(tp, &kernel_tp, sizeof (kernel_tp)))
        error = -EFAULT;

    return error;
}
```

```
which gets the reference to the desired clock through "clockid_to_kclock":
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/posix-timers.c#L1027
```

```
static structk_clock *clockid_to_kclock(constclockid_t id)
{
    if (id < 0)
        return (id &CLOCKFD_MASK) ==CLOCKFD ? &clock_posix_dynamic : &clock_posix_cpu;

    if (id >=MAX_CLOCKS || !posix_clocks[id].clock_getres)
        return NULL;
    return &posix_clocks[id];
}
```

POSIX Timers

```
These references are created by "posix_timers_register_clock":
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/posix-timers.c#L523
```

```
void posix_timers_register_clock(constclockid_tclock_id, structk_clock *new_clock)
{
    if ((unsigned)clock_id >=MAX_CLOCKS) {
        printk(KERN_WARNING "POSIX clock register failed for clock_id %d\n", clock_id);
        return;
    }

    if (!new_clock->clock_get) {
        printk(KERN_WARNING "POSIX clock id %d lacks clock_get()\n", clock_id);
        return;
    }
    if (!new_clock->clock_getres) {
        printk(KERN_WARNING "POSIX clock id %d lacks clock_getres()\n", clock_id);
        return;
    }
    posix_clocks[clock_id] = *new_clock;
}
```

```
invoked on startup by the function "init_posix_timers" (in the case of
CLOCK_RUNTIME, CLOCK_MONOTONIC and CLOCK_MONOTONIC_RAW):
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/posix-timers.c#L285
```

```
static__init int init_posix_timers(void)
{
    structk_clockclock_realtime = {
        .clock_getres   =posix_get_hrtimer_res,
        .clock_get  =posix_clock_realtime_get,
        .clock_set  =posix_clock_realtime_set,
        .clock_adj  =posix_clock_realtime_adj,
        .nsleep     =common_nsleep,
        .nsleep_restart =hrtimer_nanosleep_restart,
        .timer_create   =common_timer_create,
        .timer_set  =common_timer_set,
        .timer_get  =common_timer_get,
        .timer_del  =common_timer_del,
    };
```

```
    structk_clockclock_monotonic = {
        .clock_getres   =posix_get_hrtimer_res,
        .clock_get  =posix_ktime_get_ts,
        .nsleep     =common_nsleep,
        .nsleep_restart =hrtimer_nanosleep_restart,
        .timer_create   =common_timer_create,
        .timer_set  =common_timer_set,
        .timer_get  =common_timer_get,
        .timer_del  =common_timer_del,
    };
    structk_clockclock_monotonic_raw = {
        .clock_getres   =posix_get_hrtimer_res,
        .clock_get  =posix_get_monotonic_raw,
    };
    ...

    posix_timers_register_clock(CLOCK_REALTIME, &clock_realtime);
    posix_timers_register_clock(CLOCK_MONOTONIC, &clock_monotonic);
    posix_timers_register_clock(CLOCK_MONOTONIC_RAW, &clock_monotonic_raw);
    ...

    posix_timers_cache =kmem_cache_create("posix_timers_cache",
                    sizeof (structk_itimer), 0,SLAB_PANIC,
                    NULL);
    return 0;
}
```

and by the function "init_posix_cpu_timers" (in the case of CPU_PROCESS_CPUTIME_ID):
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/posix-cpu-timers.c#L1487

```
static __init int init_posix_cpu_timers(void)
{
    struct k_clock process = {
        .clock_getres   = process_cpu_clock_getres,
        .clock_get  = process_cpu_clock_get,
        .timer_create   = process_cpu_timer_create,
        .nsleep     = process_cpu_nsleep,
        .nsleep_restart = process_cpu_nsleep_restart,
    };
    struct k_clock thread = {
        .clock_getres   = thread_cpu_clock_getres,
        .clock_get  = thread_cpu_clock_get,
        .timer_create   = thread_cpu_timer_create,
    };
    struct timespec ts;

    posix_timers_register_clock(CLOCK_PROCESS_CPUTIME_ID, &process);
    posix_timers_register_clock(CLOCK_THREAD_CPUTIME_ID, &thread);

    cputime_to_timespec(cputime_one_jiffy, &ts);
    onecputick = ts.tv_nsec;
    WARN_ON(ts.tv_sec != 0);

    return 0;
}
```

The syscall then uses the getter "kc->clock_get()". By following the clock setup above, the getters of interest are:

REALTIME:
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/posix-timers.c#L206
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/timekeeping.c#L626
MONOTONIC:
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/posix-timers.c#L228
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/timekeeping.c#L778
MONOTONIC RAW:
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/posix-timers.c#L237
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/timekeeping.c#L1107
PROCESS_CPUTIME_ID
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/posix-cpu-timers.c#L1423
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/posix-cpu-timers.c#L309
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/posix-cpu-timers.c#L287
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/posix-cpu-timers.c#L262

The CLOCK_REALTIME and CLOCK_MONOTONIC clocks are identical up to shifting by the "tk->wall_to_monotonic" offset. These two are somehow adjusted by NTP. According to the documentation, CLOCK_MONOTONIC_RAW is NOT affected by NTP. To my understanding, all the seqlocks used here and in the upcoming sections are just an artifact of how synchronization is performed in the kernel, and so are not relevant for this timekeeping inquiry.

```
/* Get clock_realtime */
static int posix_clock_realtime_get(clockid_t which_clock, structtimespec *tp)
{
    ktime_get_real_ts(tp);
    return 0;
}
static inline void ktime_get_real_ts(struct timespec *ts)
{
    getnstimeofday64(ts);
}
/**
 * __getnstimeofday64 - Returns the time of day in a timespec64.
 * @ts:     pointer to the timespec to be set
 *
 * Updates the time of day in the timespec.
 * Returns 0 on success, or -ve when suspended (timespec will be undefined).
 */
int __getnstimeofday64(struct timespec64 *ts)
{
    struct timekeeper *tk = &tk_core.timekeeper;
    unsigned long seq;
    s64 nsecs = 0;

    do {
        seq = read_seqcount_begin(&tk_core.seq);
        ts->tv_sec = tk->xtime_sec;
        nsecs = timekeeping_get_ns(&tk->tkr_mono);

    } while (read_seqcount_retry(&tk_core.seq, seq));

    ts->tv_nsec = 0;
    timespec64_add_ns(ts, nsecs);

    /*
     * Do not bail out early, in case there were callers still using
     * the value, even in the face of the WARN_ON.
     */
    if (unlikely(timekeeping_suspended))
        return -EAGAIN;
    return 0;
}
```

```
/*
 * Get monotonic time for posix timers
 */
static int posix_ktime_get_ts(clockid_t which_clock, struct timespec *tp)
{
    ktime_get_ts(tp);
    return 0;
}
static inline void ktime_get_ts(struct timespec *ts)
{
```

```
    ktime_get_ts64(ts);
}
/**
 * ktime_get_ts64 - get the monotonic clock in timespec64 format
 * @ts:     pointer to timespec variable
 *
 * The function calculates the monotonic clock from the realtime
 * clock and the wall_to_monotonic offset and stores the result
 * in normalized timespec64 format in the variable pointed to by @ts.
 */
void ktime_get_ts64(struct timespec64 *ts)
{
    struct timekeeper *tk = &tk_core.timekeeper;
    struct timespec64 tomono;
    s64 nsec;
    unsigned int seq;

    WARN_ON(timekeeping_suspended);

    do {
        seq = read_seqcount_begin(&tk_core.seq);
        ts->tv_sec = tk->xtime_sec;
        nsec = timekeeping_get_ns(&tk->tkr_mono);
        tomono = tk->wall_to_monotonic;

    } while (read_seqcount_retry(&tk_core.seq, seq));

    ts->tv_sec += tomono.tv_sec;
    ts->tv_nsec = 0;
    timespec64_add_ns(ts, nsec + tomono.tv_nsec);
}
```

```
/*
 * Get monotonic-raw time for posix timers
 */
static int posix_get_monotonic_raw(clockid_t which_clock, struct timespec *tp)
{
    getrawmonotonic(tp);
    return 0;
}
static inline void getrawmonotonic(struct timespec *ts)
{
    getrawmonotonic64(ts);
}
/**
 * getrawmonotonic64 - Returns the raw monotonic time in a timespec
 * @ts:     pointer to the timespec64 to be set
 *
 * Returns the raw monotonic time (completely un-modified by ntp)
 */
void getrawmonotonic64(struct timespec64 *ts)
{
    struct timekeeper *tk = &tk_core.timekeeper;
    struct timespec64 ts64;
    unsigned long seq;
    s64 nsecs;

    do {
        seq = read_seqcount_begin(&tk_core.seq);
        nsecs = timekeeping_get_ns(&tk->tkr_raw);
        ts64 = tk->raw_time;

    } while (read_seqcount_retry(&tk_core.seq, seq));

    timespec64_add_ns(&ts64, nsecs);
    *ts = ts64;
}
```

```
#define PROCESS_CLOCK   MAKE_PROCESS_CPUCLOCK(0, CPUCLOCK_SCHED)
static int process_cpu_clock_get(const clockid_t which_clock, struct timespec *tp)
{
    return posix_cpu_clock_get(PROCESS_CLOCK, tp);
}

static int posix_cpu_clock_get(const clockid_t which_clock, struct timespec *tp)
{
    const pid_t pid = CPUCLOCK_PID(which_clock);
    int err = -EINVAL;

    if (pid == 0) {
        /*
         * Special case constant value for our own clocks.
         * We don't have to do any lookup to find ourselves.
         */
        err = posix_cpu_clock_get_task(current, which_clock, tp);
    } else {
        ...
    }

    return err;
}

static int posix_cpu_clock_get_task(struct task_struct *tsk,
                    const clockid_t which_clock,
                    struct timespec *tp)
{
    int err = -EINVAL;
    unsigned long long rtn;

    if (CPUCLOCK_PERTHREAD(which_clock)) {
        ...
    } else {
        if (tsk == current || thread_group_leader(tsk))
            err = cpu_clock_sample_group(which_clock, tsk, &rtn);
    }

    if (!err)
```

```
        sample_to_timespec(which_clock, rtn, tp);
    return err;
}

/*
 * Sample a process (thread group) clock for the given group_leader task.
 * Must be called with task sighand lock held for safe while_each_thread()
 * traversal.
 */
static int cpu_clock_sample_group(const clockid_t which_clock,
                  struct task_struct *p,
                  unsigned long long *sample)
{
    struct task_cputime cputime;

    switch (CPUCLOCK_WHICH(which_clock)) {
    default:
        return -EINVAL;
    case CPUCLOCK_PROF:
        ...
    case CPUCLOCK_VIRT:
        ...
    case CPUCLOCK_SCHED:
        thread_group_cputime(p, &cputime);
        *sample = cputime.sum_exec_runtime;
        break;
    }
    return 0;
}
```

RT/MONOTONIC (RAW): Struct Timekeeper

```
The main timekeeping data structure is "struct timekeeper", which stores pointers to
all the measurements used/mentioned in the previous sections/functions:
https://elixir.bootlin.com/linux/v4.3/source/include/linux/timekeeper_internal.h#L83
```

```
/**
 * struct timekeeper - Structure holding internal timekeeping values.
 * @tkr_mono:       The readout base structure for CLOCK_MONOTONIC
 * @tkr_raw:        The readout base structure for CLOCK_MONOTONIC_RAW
 * @xtime_sec:      Current CLOCK_REALTIME time in seconds
 * @ktime_sec:      Current CLOCK_MONOTONIC time in seconds
 * @wall_to_monotonic:  CLOCK_REALTIME to CLOCK_MONOTONIC offset
 * @offs_real:      Offset clock monotonic -> clock realtime
 * @offs_boot:      Offset clock monotonic -> clock boottime
 * @offs_tai:       Offset clock monotonic -> clock tai
 * @tai_offset:     The current UTC to TAI offset in seconds
 * @clock_was_set_seq:  The sequence number of clock was set events
 * @next_leap_ktime:    CLOCK_MONOTONIC time value of a pending leap-second
 * @raw_time:       Monotonic raw base time in timespec64 format
 * @cycle_interval: Number of clock cycles in one NTP interval
 * @xtime_interval: Number of clock shifted nano seconds in one NTP
 *          interval.
 * @xtime_remainder:    Shifted nano seconds left over when rounding
 *          @cycle_interval
 * @raw_interval:   Raw nano seconds accumulated per NTP interval.
 * @ntp_error:      Difference between accumulated time and NTP time in ntp
 *          shifted nano seconds.
 * @ntp_error_shift:    Shift conversion between clock shifted nano seconds and
 *          ntp shifted nano seconds.
 * @last_warning:   Warning ratelimiter (DEBUG_TIMEKEEPING)
 * @underflow_seen: Underflow warning flag (DEBUG_TIMEKEEPING)
 * @overflow_seen:  Overflow warning flag (DEBUG_TIMEKEEPING)
 *
 * Note: For timespec(64) based interfaces wall_to_monotonic is what
 * we need to add to xtime (or xtime corrected for sub jiffie times)
 * to get to monotonic time.  Monotonic is pegged at zero at system
 * boot time, so wall_to_monotonic will be negative, however, we will
 * ALWAYS keep the tv_nsec part positive so we can use the usual
 * normalization.
 *
 * wall_to_monotonic is moved after resume from suspend for the
 * monotonic time not to jump. We need to add total_sleep_time to
 * wall_to_monotonic to get the real boot based time offset.
 *
 * wall_to_monotonic is no longer the boot time, getboottime must be
 * used instead.
 */
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;
    ktime_t         next_leap_ktime;
    struct timespec64   raw_time;

    /* The following members are for timekeeping internal use */
    cycle_t         cycle_interval;
    u64         xtime_interval;
    s64         xtime_remainder;
    u32         raw_interval;
    /* The ntp_tick_length() value currently being used.
     * This cached copy ensures we consistently apply the tick
     * length for an entire tick, as ntp_tick_length may change
     * mid-tick, and we don't want to apply that new value to
     * the tick in progress.
     */
    u64         ntp_tick;
    /* Difference between accumulated time and NTP time in ntp
     * shifted nano seconds. */
    s64         ntp_error;
    u32         ntp_error_shift;
    u32         ntp_err_mult;
};
```

```
The two relevant members of "struct timekeeper", accessed by the three clocks
(CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_MONOTONIC_RAW) discussed here, are
defined as "struct tk_read_base"s:

    struct tk_read_base tkr_mono;
    struct tk_read_base tkr_raw;

This struct is defined in
https://elixir.bootlin.com/linux/v4.3/source/include/linux/timekeeper_internal.h#L30
```

```
/**
 * struct tk_read_base - base structure for timekeeping readout
 * @clock:  Current clocksource used for timekeeping.
 * @read:   Read function of @clock
 * @mask:   Bitmask for two's complement subtraction of non 64bit clocks
 * @cycle_last: @clock cycle value at last update
 * @mult:   (NTP adjusted) multiplier for scaled math conversion
 * @shift:  Shift value for scaled math conversion
 * @xtime_nsec: Shifted (fractional) nano seconds offset for readout
 * @base:   ktime_t (nanoseconds) base time for readout
 *
 * This struct has size 56 byte on 64 bit. Together with a seqcount it
 * occupies a single 64byte cache line.
 *
 * The struct is separate from struct timekeeper as it is also used
 * for a fast NMI safe accessors.
 */
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;
};
```

```
The "struct tk_read_base" is then accessed by the three getters via the
"timekeeping_get_ns()" function:
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/timekeeping.c#L301

Underneath, this function collects the "timekeeping_get_delta()"
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/timekeeping.c#L161
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/timekeeping_internal.h#L23
and performs a multiply & shift operation on it.

To my understanding, a "struct tk_read_base" keeps a reference to a clock (whose
details, e.g. increments/updates, are abstracted away), the value of that clock
at the last update, and the true time at that same last update (pre-shift). To get
the time from the "struct tk_read_base", the clock-delta
    (current_time - last_update_time)
is converted from ticks to the same units as the true time (via the "tkr->mult"
field), then added onto the last_update true time. The "tkr->shift" field then
controls the resolution of our clock (from whatever units to nanoseconds).

Rephrased, the time of this clock is the time at the previous update, plus the
internal-clock delta since that previous update, converted to common units. The
result is shifted to the desired resolution.
```

```
static inline s64 timekeeping_get_ns(struct tk_read_base *tkr)
{
    cycle_t delta;
    s64 nsec;

    delta = timekeeping_get_delta(tkr);

    nsec = delta * tkr->mult + tkr->xtime_nsec;
    nsec >>= tkr->shift;

    /* If arch requires, add in get_arch_timeoffset() */
    return nsec + arch_gettimeoffset();
}

static inline cycle_t timekeeping_get_delta(struct tk_read_base *tkr)
{
    struct timekeeper *tk = &tk_core.timekeeper;
    cycle_t now, last, mask, max, delta;
    unsigned int seq;

    /*
     * Since we're called holding a seqlock, the data may shift
     * under us while we're doing the calculation. This can cause
     * false positives, since we'd note a problem but throw the
     * results away. So nest another seqlock here to atomically
     * grab the points we are checking with.
     */
    do {
        seq = read_seqcount_begin(&tk_core.seq);
        now = tkr->read(tkr->clock);
        last = tkr->cycle_last;
        mask = tkr->mask;
    } while (read_seqcount_retry(&tk_core.seq, seq));

    delta = clocksource_delta(now, last, mask);

    // Handle underflow and overflow
    ...
    return delta;
}

static inline cycle_t clocksource_delta(cycle_t now, cycle_t last, cycle_t mask)
{
    return (now - last) & mask;
}
```

```
The "struct timekeeper tk_core.timekeeper" is initialized in "timekeeping_init":
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/timekeeping.c#L1212
which is a wrapper around "tk_setup_internals":
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/timekeeping.c#L230
What is worth noting is that "tk->tkr_mono.clock" and "tk->tkr_raw.clock" both
end up referencing "clocksource_default_clock()". In other words, this is the
primary time source in this implementation.
```

```
/*
 * timekeeping_init - Initializes the clocksource and common timekeeping values
 */
void __init timekeeping_init(void)
{
    struct timekeeper *tk = &tk_core.timekeeper;
    struct clocksource *clock;

    ...
    clock = clocksource_default_clock();
    if (clock->enable)
        clock->enable(clock);
    tk_setup_internals(tk, clock);
    ...

    raw_spin_unlock_irqrestore(&timekeeper_lock, flags);
}

/**
 * tk_setup_internals - Set up internals to use clocksource clock.
 *
 * @tk:     The target timekeeper to setup.
 * @clock:      Pointer to clocksource.
 *
 * Calculates a fixed cycle/nsec interval for a given clocksource/adjustment
 * pair and interval request.
 *
 * Unless you're the timekeeping code, you should not be using this!
 */
static void tk_setup_internals(struct timekeeper *tk, struct clocksource *clock)
{
    struct clocksource *old_clock;
```

```
    old_clock = tk->tkr_mono.clock;
    tk->tkr_mono.clock = clock;
    tk->tkr_mono.read = clock->read;
    tk->tkr_mono.mask = clock->mask;
    tk->tkr_mono.cycle_last = tk->tkr_mono.read(clock);

    tk->tkr_raw.clock = clock;
    tk->tkr_raw.read = clock->read;
    tk->tkr_raw.mask = clock->mask;
    tk->tkr_raw.cycle_last = tk->tkr_mono.cycle_last;

    // NTP configuration
    ...
}
```

RT/MONOTONIC (RAW): Clocksources

The "clocksource_default_clock" is based on Jiffies:
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/jiffies.c#L103
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/jiffies.c#L67
This is the default choice of time source in the Linux kernel. As a self-reminder, the Jiffies counter is incremented once per timer interrupt.

Note that Jiffies is initialized with a rating of 1, which is the lowest valid rating assignable to a timesource. The clocksource used in the end is the one with highest rating.

```
struct clocksource * __init __weak clocksource_default_clock(void)
{
    return &clocksource_jiffies;
}

static struct clocksource clocksource_jiffies = {
    .name       = "jiffies",
    .rating     = 1, /* lowest valid rating*/
    .read       = jiffies_read,
    .mask       = 0xffffffff, /*32bits*/
    .mult       = NSEC_PER_JIFFY << JIFFIES_SHIFT, /* details above */
    .shift      = JIFFIES_SHIFT,
    .max_cycles = 10,
};
```

However, the clocksource used within "struct timekeeper tk_core.timekeeper" can (and very likely will) be overwritten via the "change_clocksource" call
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/timekeeping.c#L1067
That is to say the default clocksource is important, but unlikely to be used for timekeeping purposes after boot has finished and other, higher-score clocksources have been initialized.

"change_clocksource" is ultimately invoked by "clocksource_select":
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/clocksource.c#L596
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/clocksource.c#L582
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/clocksource.c#L528
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/timekeeping.c#L1090
https://elixir.bootlin.com/linux/v4.3/source/kernel/stop_machine.c#L551
(these links are a sort of depth-first view of the call stack leading up to the invokation of "change_clocksource", starting from "clocksource_select"; the implementation details are not too relevant here, but still interesting).
Lastly, this is invoked from "__clocksource_register_scale":
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/clocksource.c#L739
which is where new clocksources pass through at registration in the kernel.

```
/**
 * change_clocksource - Swaps clocksources if a new one is available
 *
 * Accumulates current time interval and initializes new clocksource
 */
static int change_clocksource(void *data)
{
    struct timekeeper *tk = &tk_core.timekeeper;
    struct clocksource *new, *old;
    unsigned long flags;

    new = (struct clocksource *) data;

    raw_spin_lock_irqsave(&timekeeper_lock, flags);
    write_seqcount_begin(&tk_core.seq);

    timekeeping_forward_now(tk);
    /*
     * If the cs is in module, get a module reference. Succeeds
     * for built-in code (owner == NULL) as well.
     */
    if (try_module_get(new->owner)) {
        if (!new->enable || new->enable(new) == 0) {
            old = tk->tkr_mono.clock;
            tk_setup_internals(tk, new);
            if (old->disable)
                old->disable(old);
            module_put(old->owner);
        } else {
            module_put(new->owner);
        }
    }
    timekeeping_update(tk, TK_CLEAR_NTP | TK_MIRROR | TK_CLOCK_WAS_SET);

    write_seqcount_end(&tk_core.seq);
    raw_spin_unlock_irqrestore(&timekeeper_lock, flags);

    return 0;
}
```

The Linux Kernel exposes three functions to register a new clocksource:
https://elixir.bootlin.com/linux/v4.3/source/include/linux/clocksource.h#L208
https://elixir.bootlin.com/linux/v4.3/source/include/linux/clocksource.h#L213
https://elixir.bootlin.com/linux/v4.3/source/include/linux/clocksource.h#L218

The first of them is used solely by the Jiffies clocksource (which is the default clocksource anyway, as per the last section, so this is not interesting):
https://elixir.bootlin.com/linux/v4.3/source/kernel/time/jiffies.c#L96

The second of them (to the best of my understanding) does not register clocks at a higher rating than the TSC clocksource on x86 (see next section). It is used by the HPET clock (if available), which registers itself with a 250 rating, making it lower priority than the TSC:
https://elixir.bootlin.com/linux/v4.3/source/arch/x86/kernel/hpet.c#L768

Worth noting is that the KVM clock also registers itself via this call
https://elixir.bootlin.com/linux/v4.3/source/arch/x86/kernel/kvmclock.c#L148
with a rating of 400. This makes it the highest-rated clocksource when it is used. However, this clock is used primarily for virtualization, and so falls beyond our fingerprinting attempts.

```
static inline int __clocksource_register(struct clocksource *cs)
{
    return __clocksource_register_scale(cs, 1, 0);
}

static inline int clocksource_register_hz(struct clocksource *cs, u32 hz)
{
    return __clocksource_register_scale(cs, 1, hz);
}

static inline int clocksource_register_khz(struct clocksource *cs, u32 khz)
{
    return __clocksource_register_scale(cs, 1000, khz);
}

static struct clocksource clocksource_hpet = {
    .name       = "hpet",
    .rating     = 250,
    .read       = read_hpet,
    .mask       = HPET_MASK,
    .flags      = CLOCK_SOURCE_IS_CONTINUOUS,
    .resume     = hpet_resume_counter,
    .archdata   = { .vclock_mode = VCLOCK_HPET },
};

static struct clocksource kvm_clock = {
    .name = "kvm-clock",
    .read = kvm_clock_get_cycles,
    .rating = 400,
    .mask = CLOCKSOURCE_MASK(64),
```

```
    .flags = CLOCK_SOURCE_IS_CONTINUOUS,
};
```

The third registration method ("clocksource_register_khz") is used by the TSC counter (supposedly at or around boot time), which involves a quick calibration:
https://elixir.bootlin.com/linux/v4.3/source/arch/x86/kernel/tsc.c#L1140
This call schedules a delayed, longer calibration of a refined TSC clocksource ("tsc_refine_calibration_work") via the "tsc_irqwork" variable:
https://elixir.bootlin.com/linux/v4.3/source/arch/x86/kernel/tsc.c#L1082
which might or might not take effect, depending on whether the calibration results align with the quick boot-time TSC calibration. Nevertheless, the TSC-based clocksource is registered this way. The definition of the TSC clocksource is:
https://elixir.bootlin.com/linux/v4.3/source/arch/x86/kernel/tsc.c#L986

The "clocksource_tsc" is initialized with a rating of 300 [opinion: and, on a system which increments the TSC counter at a fixed rate, which includes most of the latest systems, this rating is unlikely to be downgraded]. This leads me to the conclusion that all of CLOCK_REALTIME, CLOCK_MONOTONIC and CLOCK_MONOTONIC_RAW are based on the TSC counter.

```
static struct clocksource clocksource_tsc = {
    .name                   = "tsc",
    .rating                 = 300,
    .read                   = read_tsc,
    .mask                   = CLOCKSOURCE_MASK(64),
    .flags                  = CLOCK_SOURCE_IS_CONTINUOUS |
                  CLOCK_SOURCE_MUST_VERIFY,
    .archdata               = { .vclock_mode = VCLOCK_TSC },
};

static DECLARE_DELAYED_WORK(tsc_irqwork, tsc_refine_calibration_work);
static int __init init_tsc_clocksource(void)
{
    if (!cpu_has_tsc || tsc_disabled > 0 || !tsc_khz)
        return 0;

    if (tsc_clocksource_reliable)
        clocksource_tsc.flags &= ~CLOCK_SOURCE_MUST_VERIFY;
    /* lower the rating if we already know its unstable: */
    if (check_tsc_unstable()) {
        clocksource_tsc.rating = 0;
        clocksource_tsc.flags &= ~CLOCK_SOURCE_IS_CONTINUOUS;
    }

    if (boot_cpu_has(X86_FEATURE_NONSTOP_TSC_S3))
        clocksource_tsc.flags |= CLOCK_SOURCE_SUSPEND_NONSTOP;

    /*
     * Trust the results of the earlier calibration on systems
     * exporting a reliable TSC.
     */
    if (boot_cpu_has(X86_FEATURE_TSC_RELIABLE)) {
        clocksource_register_khz(&clocksource_tsc, tsc_khz);
        return 0;
    }

    schedule_delayed_work(&tsc_irqwork, 0);
    return 0;
}
```

PROCESS_CPUTIME_ID: Scheduler

As seen above, the CLOCK_PROCESS_CPUTIME_ID getter invokes "thread_group_cputime":
https://elixir.bootlin.com/linux/v4.3/source/kernel/sched/cputime.c#L286

```
/*
 * Accumulate raw cputime values of dead tasks (sig->[us]time) and live
 * tasks (sum on group iteration) belonging to @tsk's group.
 */
void thread_group_cputime(struct task_struct *tsk, struct task_cputime *times)
{
    struct signal_struct *sig = tsk->signal;
    cputime_t utime, stime;
    struct task_struct *t;
    unsigned int seq, nextseq;
    unsigned long flags;

    rcu_read_lock();
    /* Attempt a lockless read on the first round. */
    nextseq = 0;
    do {
        seq = nextseq;
        flags = read_seqbegin_or_lock_irqsave(&sig->stats_lock, &seq);
        times->utime = sig->utime;
        times->stime = sig->stime;
        times->sum_exec_runtime = sig->sum_sched_runtime;

        for_each_thread(tsk, t) {
            task_cputime(t, &utime, &stime);
            times->utime += utime;
            times->stime += stime;
            times->sum_exec_runtime += task_sched_runtime(t);
        }
        /* If lockless access failed, take the lock. */
        nextseq = 1;
    } while (need_seqretry(&sig->stats_lock, seq));
    done_seqretry_irqrestore(&sig->stats_lock, seq, flags);
    rcu_read_unlock();
}
```

In turn, this computes "sum_exec_runtime" (relevant for our measurements) through calls to "task_sched_runime(t)":
https://elixir.bootlin.com/linux/v4.3/source/kernel/sched/core.c#L2771
This quantity is managed by the Linux scheduler.

```
/*
 * Return accounted runtime for the task.
 * In case the task is currently running, return the runtime plus current's
 * pending runtime that have not been accounted yet.
 */
unsigned long long task_sched_runtime(struct task_struct *p)
{
    unsigned long flags;
    struct rq *rq;
    u64 ns;

#if defined(CONFIG_64BIT) && defined(CONFIG_SMP)
    /*
     * 64-bit doesn't need locks to atomically read a 64bit value.
     * So we have a optimization chance when the task's delta_exec is 0.
     * Reading ->on_cpu is racy, but this is ok.
     *
     * If we race with it leaving cpu, we'll take a lock. So we're correct.
     * If we race with it entering cpu, unaccounted time is 0. This is
     * indistinguishable from the read occurring a few cycles earlier.
     * If we see ->on_cpu without ->on_rq, the task is leaving, and has
     * been accounted, so we're correct here as well.
     */
    if (!p->on_cpu || !task_on_rq_queued(p))
        return p->se.sum_exec_runtime;
```

```
#endif

    rq = task_rq_lock(p, &flags);
    /*
     * Must be ->curr _and_ ->on_rq.  If dequeued, we would
     * project cycles that may never be accounted to this
     * thread, breaking clock_gettime().
     */
    if (task_current(rq, p) && task_on_rq_queued(p)) {
        update_rq_clock(rq);
        p->sched_class->update_curr(rq);
    }
    ns = p->se.sum_exec_runtime;
    task_rq_unlock(rq, p, &flags);

    return ns;
}
```

The important (clock-related) step in the code above is the call to the function "update_rq_clock":
https://elixir.bootlin.com/linux/v4.3/source/kernel/sched/core.c#L98
which calls "sched_clock_cpu":
https://elixir.bootlin.com/linux/v4.3/source/kernel/sched/clock.c#L294
Finally, (assuming the scheduler clock is stable), this is a wrapper around the "sched_clock" function.

```
void update_rq_clock(struct rq *rq)
{
    s64 delta;

    lockdep_assert_held(&rq->lock);

    if (rq->clock_skip_update & RQCF_ACT_SKIP)
        return;

    delta = sched_clock_cpu(cpu_of(rq)) - rq->clock;
    if (delta < 0)
        return;
    rq->clock += delta;
    update_rq_clock_task(rq, delta);
}

u64 sched_clock_cpu(int cpu)
{
    struct sched_clock_data *scd;
    u64 clock;

    if (sched_clock_stable())
        return sched_clock();

    if (unlikely(!sched_clock_running))
        return 0ull;

    preempt_disable_notrace();
    scd = cpu_sdc(cpu);

    if (cpu != smp_processor_id())
        clock = sched_clock_remote(scd);
    else
        clock = sched_clock_local(scd);
    preempt_enable_notrace();

    return clock;
}
```

The situation is now very similar to the REALTIME/MONOTONIC clocks. The default "sched_clock" implementation
https://elixir.bootlin.com/linux/v4.3/source/kernel/sched/clock.c#L70
is based on Jiffies, and is expected to be overwritten later on by some more accurate timesource.

https://elixir.bootlin.com/linux/v4.3/source/arch/x86/kernel/tsc.c#L309
https://elixir.bootlin.com/linux/v4.3/source/arch/x86/include/asm/paravirt.h#L181
https://elixir.bootlin.com/linux/v4.3/source/arch/x86/kernel/paravirt.c#L328
https://elixir.bootlin.com/linux/v4.3/source/arch/x86/kernel/tsc.c#L276

```
/*
 * Scheduler clock - returns current time in nanosec units.
 * This is default implementation.
 * Architectures and sub-architectures can override this.
 */
unsigned long long __weak sched_clock(void)
{
    return (unsigned long long)(jiffies - INITIAL_JIFFIES)
                    * (NSEC_PER_SEC / HZ);
}
```

On x86, this ends up being the case when the TSC overwrites "sched_clock":
https://elixir.bootlin.com/linux/v4.3/source/arch/x86/kernel/tsc.c#L309
https://elixir.bootlin.com/linux/v4.3/source/arch/x86/include/asm/paravirt.h#L181
https://elixir.bootlin.com/linux/v4.3/source/arch/x86/kernel/paravirt.c#L328
https://elixir.bootlin.com/linux/v4.3/source/arch/x86/kernel/tsc.c#L276
So, unless something about the TSC is misconfigured or misbehaving, this clock (the scheduler clock and CLOCK_PROCESS_CPUTIME_ID) will also be based on the TSC.

```
unsigned long long sched_clock(void)
{
    return paravirt_sched_clock();
}

struct pv_time_ops pv_time_ops = {
    .sched_clock = native_sched_clock,
    .steal_clock = native_steal_clock,
};
static inline unsigned long long paravirt_sched_clock(void)
{
    return PVOP_CALL0(unsigned long long, pv_time_ops.sched_clock);
}

/*
 * Scheduler clock - returns current time in nanosec units.
 */
u64 native_sched_clock(void)
{
    if (static_branch_likely(&__use_tsc)) {
        u64 tsc_now = rdtsc();

        /* return the value in ns */
        return cycles_2_ns(tsc_now);
    }

    /*
     * Fall back to jiffies if there's no TSC available:
     * ( But note that we still use it if the TSC is marked
     *   unstable. We do this because unlike Time Of Day,
     *   the scheduler clock tolerates small errors and it's
     *   very important for it to be as fast as the platform
     *   can achieve it. )
     */

    /* No locking but a rare wrong value is not a big deal: */
```

```
    return (jiffies_64 - INITIAL_JIFFIES) * (1000000000 / HZ);
}
```