Time Management

Have a look

- Before getting into **Time Management**, please take a look at
  - LK_Bird's Eye View session,

    S7 Process Scheduler,Time,IPC.

Time Management

- General Timing Hardware
  - Real-time Clock **(RTC)**
    Keeping track of the current time, date and use it as a timestamp for various resources in the system.

  - Timestamp Counter **(TSC)**
    Implemented in every x86 microprocessor by means of a 64-bit register called **TSC**.

    It counts the number of clock signals arriving on the CLK pin of the processor as the fact that one processor clock's frequency might not be the same as others makes it vary across processors.

    native_calibrate_tsc() : CPU clock frequency is calculated during system boot.

/arch/x86/kernel/tsc.c

```c
/**
 * native_calibrate_tsc
 * Determine TSC frequency via CPUID, else return 0.
 */
unsigned long native_calibrate_tsc(void)
{
    ...
}
```

Time Management

- General Timing Hardware
  - Programmable Interrupt Timer **(PIT)**
    tasks that need to be carried out by the kernel at regular intervals.

    In most machines the PIT keeps on issuing timer interrupts on IRQ0 periodically at approximately **100 Hz** frequency **(tick rate)** leaving more time for user-mode code (programs) to execute without interruption.

  - High-precision event timer **(HPET)**
    The HPET works with clock signals in excess of 10 Mhz, issuing interrupts once every **100 ns.**

Time Management

- Hardware abstraction
  - Every system has at least one clock counter which is managed by a kernel structure.
  - HW abstraction is provided by ***struct clocksource***, this is the structure used for system time.
  - This structure provides callbacks to access and control this clock source ***read(), enable(), disable(), suspend, and resume*** routines.

/include/linux/clocksource.h

```
struct clocksource {
    u64         (*read)(struct clocksource *cs);
    u64         mask;
    u32         mult;
    u32         shift;
    u64         max_idle_ns;
    u32         maxadj;
    u32         uncertainty_margin;
#ifdef CONFIG_ARCH_CLOCKSOURCE_DATA
    struct arch_clocksource_data archdata;
#endif
    u64         max_cycles;
    const char      *name;
    struct list_head    list;
    int         rating;
    enum clocksource_ids    id;
    enum vdso_clock_mode    vdso_clock_mode;
    unsigned long       flags;

    int         (*enable)(struct clocksource *cs);
    void            (*disable)(struct clocksource *cs);
    void            (*suspend)(struct clocksource *cs);
    void            (*resume)(struct clocksource *cs);
    void            (*mark_unstable)(struct clocksource *cs);
    void            (*tick_stable)(struct clocksource *cs);

    /* private: */
#ifdef CONFIG_CLOCKSOURCE_WATCHDOG
    /* Watchdog related data, used by the framework */
    struct list_head    wd_list;
    u64         cs_last;
    u64         wd_last;
#endif
    struct module       *owner;
};
```

Time Management

- Linux Timekeeping
  - ***jiffies***
    32-bit variable holds the number of ticks elapsed since system bootup.

    jiffies_64 is used instead, which allows for thousands of millions of years before the overflow occurs.

    Some APIs :
    get_jiffies_64() : get the current value of jiffies.

    time_after(), time_before(), time_after_eq(), time_before_eq() : time comparison taking into account the possibility of wraparound.

    jiffies_to_msecs(), jiffies_to_usecs(), jiffies_to_nsecs() : convert jiffies to other time units such as ms, us, ns.

/kernel/time/jiffies.c

```c
u64 get_jiffies_64(void)
{
    unsigned int seq;
    u64 ret;

    do {
        seq = read_seqcount_begin(&jiffies_seq);
        ret = jiffies_64;
    } while (read_seqcount_retry(&jiffies_seq, seq));
    return ret;
}
```

/kernel/time/time.c

```c
#define time_after(a,b)     \
    (typecheck(unsigned long, a) && \
     typecheck(unsigned long, b) && \
     ((long)((b) - (a)) < 0))
#define time_before(a,b)    time_after(b,a)

#define time_after_eq(a,b)  \
    (typecheck(unsigned long, a) && \
     typecheck(unsigned long, b) && \
     ((long)((a) - (b)) >= 0))
#define time_before_eq(a,b) time_after_eq(b,a)
```

/include/linux/jiffies.h

```c
unsigned int jiffies_to_msecs(const unsigned long j)
{
#if HZ <= MSEC_PER_SEC && !(MSEC_PER_SEC % HZ)
    return (MSEC_PER_SEC / HZ) * j;
#elif HZ > MSEC_PER_SEC && !(HZ % MSEC_PER_SEC)
    return (j + (HZ / MSEC_PER_SEC) - 1)/(HZ / MSEC_PER_SEC);
#else
# if BITS_PER_LONG == 32
    return (HZ_TO_MSEC_MUL32 * j + (1ULL << HZ_TO_MSEC_SHR32) - 1) >>
           HZ_TO_MSEC_SHR32;
# else
    return DIV_ROUND_UP(j * HZ_TO_MSEC_NUM, HZ_TO_MSEC_DEN);
# endif
#endif
}
```

Time Management

- Linux Timekeeping, *Cont'd*
  - Timeval and timespec
    this current time is maintained by keeping the number of seconds elapsed since midnight of January 01, 1970 (called ***epoch***).

    The second elements in each of these represent the time elapsed since the last second in **us** and **ns** respectively ***timespec, timeval.***

    So, the time (counter value) read from the clock source needs to be accumulated and tracked.

    The structures ***tk_read_base, timekeeper*** serve this purpose.

/include/uapi/linux/time.h

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

struct timeval {
    __kernel_old_time_t tv_sec;     /* seconds */
    __kernel_suseconds_t    tv_usec;    /* microseconds */
};
```

/include/linux/timekeeper_internal.h

```
struct tk_read_base {
    struct clocksource  *clock;
    u64         mask;
    u64         cycle_last;
    u32         mult;
    u32         shift;
    u64         xtime_nsec;
    ktime_t         base;
    u64         base_real;
};

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 Management

- Timers
  - Timers sometimes called dynamic timers or kernel timers.
  - Timers are represented by struct ***timer_list.***
  - Steps using timers
    1) Define a timer
    i.e. ***struct timer_list my_timer;***

    2) Fill out internal values (expires, callback)
    i.e. ***my_timer.expires = jiffies + delay;***
    i.e. ***my_timer.function = my_function;***

    3) Activate the timer
    i.e. ***add_timer(&my_timer);***
  - Other APIs
    mod_timer() : changes the expiration of a given timer

    del_timer() : deactivate a timer prior to its expiration, not wait for handler execution on other CPUs.
    del_timer_sync() : deactivate a timer and wait for the handler to finish.

include/linux/timer.h

```
struct timer_list {
    struct hlist_node   entry;
    unsigned long       expires;
    void            (*function)(struct timer_list *);
    u32         flags;

#ifdef CONFIG_LOCKDEP
    struct lockdep_map  lockdep_map;
#endif
};
```

/kernel/time/timer.c

```
int mod_timer(struct timer_list *timer, unsigned long expires)
{
    ...
}
EXPORT_SYMBOL(mod_timer);

int del_timer(struct timer_list *timer)
{
    ...
}
EXPORT_SYMBOL(del_timer);

int try_to_del_timer_sync(struct timer_list *timer)
{
    ...
}
EXPORT_SYMBOL(try_to_del_timer_sync);
```

Time Management

- Timers*, Cont'd*
  - Small Delays
    void udelay(unsigned long usecs);
    void ndelay(unsigned long nsecs);
    void mdelay(unsigned long msecs);