Mentoring Operating System (MentOS)

Software Timers

Created by
Filippo Ziche, Daniele Nicoletti
Enrico Fraccaroli
enrico.fraccaroli@gmail.com

Table of Contents

Introduction

Introduction

Software Timers

Software Timers

Definition

A timer is a software facility that allows functions to be invoked at some future moment, after a given time interval has elapsed; a time-out denotes a moment at which the time interval associated with a timer has elapsed.

Timers are widely used both by the kernel and by processes. Most device drivers use timers to detect anomalous conditions—floppy disk drivers, for instance, use timers to switch off the device motor after the floppy has not been accessed for a while, and parallel printer drivers use them to detect erroneous printer conditions.

Timers are also used quite often by programmers to force the execution of specific functions at some future time (see for example the `setitimer` and `alarm` System Calls).

Introduction

Clock and Ticks system

Clock and Ticks system

At the heart of the operating system there is a clock.

Every time it cycles the `timer_handler` function is called. This function runs the scheduler but also increments a variable called `timer_ticks` which is used to keep track of the amount of time passed since the start of the system.

```
void timer_handler(pt_regs *reg) {
    ...

    // Check if a second has passed.
    timer_seconds += ((++timer_ticks % TICKS_PER_SECOND) == 0);
    // Update all timers
    run_timer_softirq();
    // Perform the schedule.
    scheduler_run(reg);

    ...
}
```

Clock and Ticks system

Every timers has an `expires` field, it contains the amount of ticks needed for it to expire. If we want, for example, a timers with a time-out of 4 seconds we can add the correspondent amount of ticks (4000 in MentOS) to the current `timer_ticks` value.

```
int seconds = 4:
sleep_timer->expires  = timer_get_ticks() + TICKS_PER_SECOND * seconds;
```

At every clock cycle we check if the `expires` field is less or equal the current `timer_ticks` and in that case we invoked the delayed function associated with the timer.

MentOS

MentOS

Dynamic Timers

Dynamic Timers

In MentOS the software timers are called Dynamic Timers, they are dynamically created and destroyed. No limit is placed on the number of currently active dynamic timers.

```
struct timer_list {
    /// Protects the access to the timer.
    spinlock_t lock;
    /// Lists of timers are mantained using the list_head.
    struct list_head entry;
    /// Ticks value when the timer has to expire
    unsigned long expires;
    /// Functions to be executed when the timer expires
    void (*function)(unsigned long);
    /// Custom data to be passed to the timer function
    unsigned long data;
    /// Pointer to the structure containing all the other related timers.
    tvec_base_t *base;
};
```

Dynamic Timers

We can initialize, add and remove timers using the following functions inside `inc/hardware/timer.h`

```
/// Initializes a new timer struct.
void init_timer(struct timer_list *timer);

/// Add a new timer to the current CPU.
void add_timer(struct timer_list *timer);

/// Removes a timer from the current CPU.
void del_timer(struct timer_list *timer);
```

Note

Timers are stored per-CPU: every CPU manages a subset of timers.

MentOS

Hierarchical Timing Wheels

Hierarchical Timing Wheels

The performance of software timers is critical. The `timer_handler` function has to check all timers and determine if they have expired and then execute a context switch. It has to be as fast as possible, if we have a lot of timers (think of a server handling socket connections for thousands of users) a slow data structure will grind the system to a halt.

There exists multiple data structures for storing the timers:

- Unordered Timer List
- Ordered Timer List
- Timer Trees
- Simple Timing Wheels
- Hashing Wheel with Ordered Timer Lists
- Hierarchical Timing Wheels

Hierarchical Timing Wheels

It this slides we will describe only the currently implemented system in MentOS: Hierarchical Timing Wheels.

The adopted solution is based on a clever data structure that partitions the expires values into blocks of ticks and allows dynamic timers to percolate efficiently from lists with larger expires values to lists with smaller ones.

In other words, instead of storing every timer in a single list we distribute the timers in multiple arrays called `timer_vec`. Each `timer_vec` is a ring buffer of lists of timers.

This allows amortised O(1) time complexity for all operations: *insert*, *delete*, *update*.

Hierarchical Timing Wheels

The timers are stored inside the `tvec_base_s` structure.

```
typedef struct tvec_base_s {
    /// The earliest expiration time of the dynamic timers yet to be checked
    unsigned long timer_ticks;
    /// Lists of timers that will expires in the next 255 ticks
    struct timer_vec_root tv1;
    /// Lists of timers that will expires in the next 2^14 - 1 ticks
    struct timer_vec tv2;
    /// Lists of timers that will expires in the next 2^20 - 1 ticks
    struct timer_vec tv3;
    /// Lists of timers that will expires in the next 2^26 - 1 ticks
    struct timer_vec tv4;
    /// Lists of timers with extremely large expires fields (2^32 - 1 ticks)
    struct timer_vec tv5;

} tvec_base_t;
```

Hierarchical Timing Wheels

The tv1 field is a structure of type timer_vec_root, which includes a vec array of 256 list_head elements, i.e., lists of dynamic timers.

```
struct timer_vec_root {
    struct list_head vec[TVR_SIZE];
};
```

The tv2, tv3, and tv4 fields are structures of type timer_vec consisting of an array caled vec of 64 list_head elements. These lists contain all dynamic timers that will decay within the next 2^{14}˘1, 2^{20}˘1, $2^{26} - 1$ ticks, respectively.

```
struct timer_vec {
    struct list_head vec[TVN_SIZE];
}
```

The tv5 field is identical to the previous ones, except that the last entry of the internal array vec is a list that includes dynamic timers with extremely large expires fields.

Hierarchical Timing Wheels

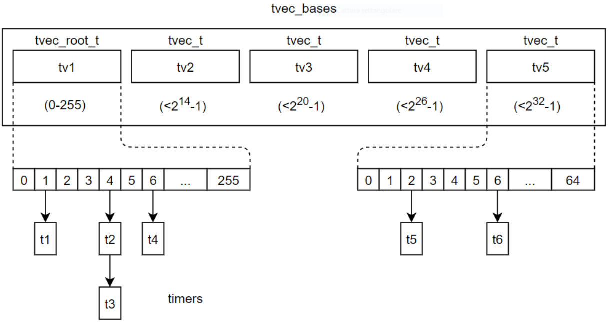


Figure: Diagram of `tvec_bases` structure

Hierarchical Timing Wheels

The `tv1` array is indexed directly by the bottom bits of the `timer_ticks` value to find the next set of events to execute.

When the kernel has, over the course of 256 ticks, cycled through the entire `tv1` array, that array must be replenished with the next 256 ticks worth of events. Replenishing is done by using the next set of ticks bits (six, normally) to index into the next array `tv2`, which points to those 256 ticks of timer entries.

Those entries are "cascaded" down to `tv1` and distributed into the appropriate slots depending on their expiration times. When `tv2` is exhausted, it is replenished from `tv3` in the same way. This process continues up to `tv5`.

Hierarchical Timing Wheels

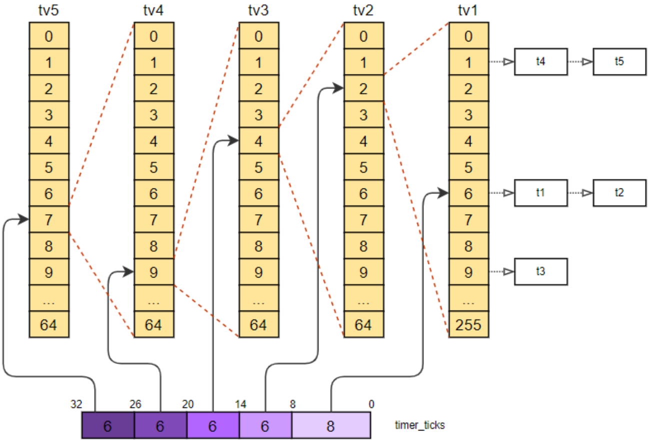


Figure: How the bits of the `timer_ticks` variable are used to index the various wheels

Hierarchical Timing Wheels

How the cascade logic inside `run_timer_softirq` works:

```
// Index of the current timer to execute
int current_time_index = base->timer_ticks & TVR_MASK;

// If the index is zero then all lists in tv1 have been checked and are empty
if (!current_time_index) {
    int tv2_index = (base->timer_ticks >> TIMER_TICKS_BITS(0)) & TVN_MASK;
    int tv3_index = (base->timer_ticks >> TIMER_TICKS_BITS(1)) & TVN_MASK;
    int tv4_index = (base->timer_ticks >> TIMER_TICKS_BITS(2)) & TVN_MASK;
    int tv5_index = (base->timer_ticks >> TIMER_TICKS_BITS(3)) & TVN_MASK;

    // Cascade timers up in the hierarchy
    if (!cascade(base, &base->tv2, tv2_index, 2) &&
        !cascade(base, &base->tv3, tv3_index, 3) &&
        !cascade(base, &base->tv4, tv4_index, 4) &&
        !cascade(base, &base->tv5, tv5_index, 5));
}
```

Hierarchical Timing Wheels

The `cascade` function removes the timers from the current time slot and reinserts them inside the data structure:

```
int cascade(tvec_base_t* base, timer_vec* tv, int time_index, int tv_index)
{
    if (!list_head_empty(tv->vec + time_index)) {
        // Reinsert all timers into base in the new correct list.
        struct list_head *it, *tmp;
        list_for_each_safe (it, tmp, tv->vec + time_index) {
            //
            struct timer_list *timer = list_entry(it, struct timer_list, entry);
            //
            list_head_del(it);
            //
            __add_timer_tvec_base(base, timer);
        }
    }
    return time_index;
}
```

Hierarchical Timing Wheels

The `__add_timer_tvec_base` function uses the `__find_tvec` function to find the correct wheel where to insert the timer.

We know that each wheel represents all the events that will happen in a certain amount of time, we consider the delta time between the expire field of the timer and the current `timer_ticks` and then pick the first wheel in the hierarchy that can hold that delta.

```c
unsigned long expires = timer->expires;
unsigned long ticks = expires - base->timer_ticks;

// How many ticks in the future a wheel can store
// TIMER_TICKS_BITS(N) = (TVR_BITS + TVN_BITS * (N))
// TIMER_TICKS(N)      = (1 << TIMER_TICKS_BITS(N))

unsigned long tv1_ticks = TIMER_TICKS(0); // 2^8  ticks in the future
unsigned long tv2_ticks = TIMER_TICKS(1); // 2^14 ticks in the future
unsigned long tv3_ticks = TIMER_TICKS(2); // 2^20 ticks in the future
unsigned long tv4_ticks = TIMER_TICKS(3); // 2^26 ticks in the future
```

Hierarchical Timing Wheels

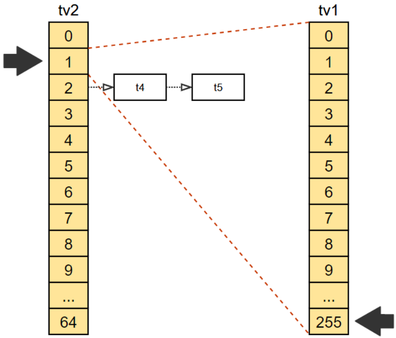
```
if (ticks < tv1_ticks) {
    *index = expires & TVR_MASK;
    *tv_index = 1;
}
else if (ticks < tv2_ticks) {
    *index = (expires >> TIMER_TICKS_BITS(0)) & TVN_MASK;
    *tv_index = 2;
}
else if (ticks < tv3_ticks) {
    *index = (expires >> TIMER_TICKS_BITS(1)) & TVN_MASK;
    *tv_index = 3;
}
else if (ticks < tv4_ticks) {
    *index = (expires >> TIMER_TICKS_BITS(2)) & TVN_MASK;
    *tv_index = 4;
}
else {
    *index = (expires >> TIMER_TICKS_BITS(3)) & TVN_MASK;
    *tv_index = 5;
}
```

MentOS

Example

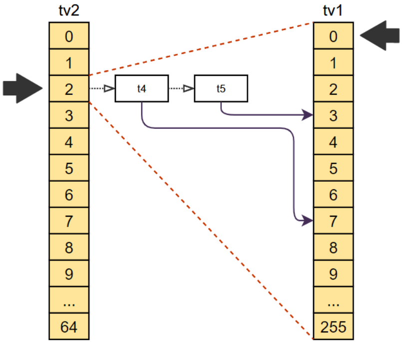
Hierarchical Timing Wheels - Simple Example

Let's look at an example. This is an initial configuration of the wheels. We have completed a cycle of `tv1` and we have two active timers, `t5` and `t4`.



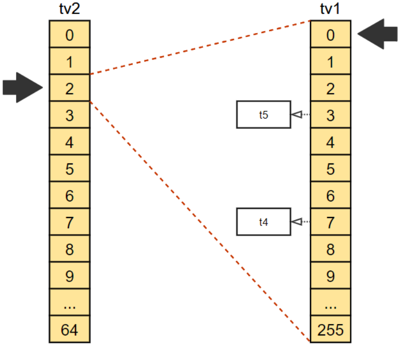
Hierarchical Timing Wheels - Simple Example

We advance the `timer_ticks` variable and the wheels. Then we relocate the timers in the correct position inside `tv1` using their expire field and `__find_tvec` function.



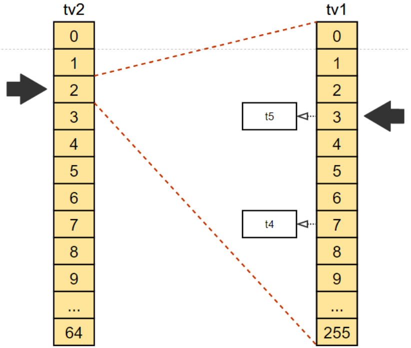
Hierarchical Timing Wheels - Simple Example

This is the state of the wheels after calling the `cascade` function.



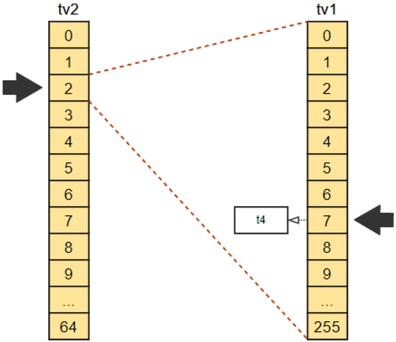
Hierarchical Timing Wheels - Simple Example

After 3 ticks, `t5` is executed and removed from the wheels.



Hierarchical Timing Wheels - Simple Example

After another 4 ticks, also `t4` expires and is consequently removed.



MentOS

Performance

Hierarchical Timing Wheels - Performance

To sum up, this rather complex algorithm ensures excellent performance. In 255 timer interrupt occurrences out of 256 (in 99.6% of the cases), the `run_timer_softirq` function just runs the functions of the decayed timers, if any.

To replenish `tv1` periodically, it is sufficient 63 times out of 64 to partition one list of `tv2` into the 256 lists of `tv1`.

The `tv2` array, in turn, must be replenished in 0.006 percent of the cases (that is, once every 16.4 seconds).

Similarly, `tv3` is replenished every 17 minutes and 28 seconds, and `tv4` is replenished every 18 hours and 38 minutes. `tv5` does not need to be replenished.