Timers

Timers play a critical role in protocol implementation.

`struct timer_list` is defined in `include/linux/timer.h`.

```
10 struct timer_list {
11    struct list_head entry;
12    unsigned long expires;
13
14    void (*function)(unsigned long);
15    unsigned long data;
16
17    struct tvec_t_base_s *base;
18 }
```

Functions of structure elements:

- **expires**: Desired expiration time of timer in *jiffies*. This is an absolute time not a relative time!
- **function**: This function is called with `data` as its argument, when timer expires.
- **data**: Typically this is a `void *` pointer used to pass the `sock` structure to the timer handler.
- **entry**: The `next` pointer of this list header will be NULL if an only if the timer is presently *disarmed*. 
The *init_timer()* function.

All timers must be initialized *before* any other timer calls can be used. The *next* pointer of the list header is set to NULL to indicate that the timer is presently *disarmed*.

```c
139 /***
140  * init_timer - initialize a timer.
141  * @timer: the timer to be initialized
142  *
143  * init_timer() must be done to a timer prior calling *any* of
144  * the other timer functions.
145 */
146 voidfastcall init_timer(struct timer_list *timer) 
147 { 
148     timer->entry.next = NULL;
149     timer->base = __raw_get_cpu_var(tvec_bases);
150 }
```
The *add_timer()* function.

This function may be used to *arm* a timer. It *must not* be invoked if the timer is already armed.

The caller of this function must ensure that the *function*, *data*, and *expires* elements are properly set up before calling *add_timer*. The *timer_pending()* function may be used to determine whether or not a timer is presently armed.

```c
66 /***
67  * add_timer - start a timer
68  * @timer: the timer to be added
69  *
70  * The kernel will do a ->function(->data) callback from the
71  * timer interrupt at the ->expires point in the future. The
72  * current time is 'jiffies'.
73  *
74  * The timer's ->expires, ->function
75  * (and if the handler uses it, ->data)
76  * fields must be set prior calling this function.
77  *
78  * Timers with an ->expires field in the past will be executed
79  * in the nexttimer tick.
80 */
81 static inline void add_timer(struct timer_list *timer)
82 {
83    BUG_ON(timer_pending(timer));
84    __mod_timer(timer, timer->expires);
85 }
```

The *timer_pending()* function

```c
54 static inline int timer_pending(const struct timer_list * timer) 55 {
56    return timer->entry.next != NULL;
57 }
```
The `mod_timer()` function.

This function may be used to arm or rearm a timer. If the timer is not armed `mod_timer()` is equivalent to `add_timer`. If the timer is already armed, the function is equivalent to `del_timer()` followed by `add_timer()`. It returns 0 if the timer was not previously armed and 1 if the timer was previously armed.

```c
258 /**
259  * mod_timer - modify a timer's timeout
260  * @timer: the timer to be modified
261  *
262  * mod_timer is a more efficient way to update the expire field of an
263  * active timer (if the timer is inactive it will be activated)
264  *
265  * mod_timer(timer, expires) is equivalent to:
266  *
267  *     del_timer(timer); timer->expires = expires;
268  *       add_timer(timer);
269  *
270  * Note that if there are multiple unserialized concurrent users of the
271  * same timer, then mod_timer() is the only safe way to modify the timeout,
272  *
273  * The function returns whether it has modified a pending timer or not.
274  * (ie. mod_timer() of an inactive timer returns 0, mod_timer()
275  * of an
276  */
```
The `mod_timer()` function

The `mod_timer()` function verifies that a handler has been supplied.

```c
277 int mod_timer(struct timer_list *timer, unsigned long expires) {
278    BUG_ON(!timer->function);
279    /*
280     * This is a common optimization triggered by the
281     * networking code - if the timer is re-modified
282     * to be the same thing then just return:
283     */
284    if (timer->expires == expires && timer_pending(timer))
285       return 1;
286    return __mod_timer(timer, expires);
287 }
```
The \textit{del_timer()} function.

This function can be used to disarm a previously armed timer. It may also be called on a timer that is not presently armed. It returns 1 if the timer was armed and 0 if it was not.

\begin{verbatim}
/*
 * del_timer - deactivate a timer.
 * @timer: the timer to be deactivated
 * del_timer() deactivates a timer -
 * this works on both active and inactive timers.
 * The function returns whether
 * it has deactivated a pending timer or not.
 * (ie. del_timer() of an inactive timer returns 0,
 * del_timer() of an active timer returns 1.)
 */

int del_timer(struct timer_list *timer)
{
    tvec_base_t *base;
    unsigned long flags;
    int ret = 0;

    if (timer_pending(timer)) {
        base = lock_timer_base(timer, &flags);
        if (timer_pending(timer)) {
            detach_timer(timer, 1);
            ret = 1;
        }
        spin_unlock_irqrestore(&base->lock, flags);
    }
    return ret;
}
\end{verbatim}
Use of timers in networking

Two approaches are commonly used for timing in networks.

*Ad hoc timer* -- A timer is associated with a specific occurrence (e.g. receipt of an ACK for a connection request or packet.

*Periodic timer* - A timer expires on a periodic basis and the timer handler is responsible for re-arming it.

Relative advantages and disadvantages of the two approaches include:

+ Ad hoc timers provide more precise timing services
- An excessive number of ad hoc timers adds timer processing overhead
- Periodic timers provide tick overhead even when no timed event has occurred.
+ Periodic timers may process multiple events in a single dispatch

Examples -

Initializing and arming a timer:

```c
init_timer(&(cpo->resend_timer));
cpo->resend_timer.function = cop_timeout;
cpo->resend_timer.data = (void *)sk;
mod_timer(&cpo->resend_timer, jiffies + RESEND_DELAY);
```
void cop_timeout(
    struct sock *sk)
{
    cop_sock_t     *cpo = (cop_sock_t *)sk;
    cop_oqe_t      *oqe;
    unsigned char   nxtack;

    /* Make sure this sock is alive and support acknowledged datagram */
    /* service                                                        */
    if ((cpo == NULL) || !(cpo->state & CPS_AD))
        goto done;

    if (cpo->timeouts >= COP_MAX_TIMEOUTS)
    {
        printk("cop_timer: SHUTDOWN! timeouts is %lu \n", cpo->timeouts);
        sk->sk_shutdown |= SEND_SHUTDOWN;
        if (sk->sk_sleep && waitqueue_active(sk->sk_sleep))
            wake_up_interruptible(sk->sk_sleep);
        return;
    }

    done:
        mod_timer(&cpo->resend_timer, jiffies + RESEND_DELAY);

    If there exists any unacked packets
    {
        locate the oqe associated with cop->lastack
        if ((present time - ttime) > RESEND_DELAY)
        {
            perform gobackN
            increment cop->timeouts // Reset this on each good (non-dup) ack
        }
    }
}