KERNEL TIMERS

Harin Chandu
rharin@cdac.in

Timer in linux kernel

1. In Linux, Kernel keeps track on the flow of time by timer interrupts.
2. The timer interrupts generated at regular timer intervals by using system Hardware.
3. The value of internal kernel counter increases for every timer interrupt.
4. “0” - system boots up & number of clocktics - since the last boot

Uses of Kernel Timers

- Timers are used to schedule the execution of a function at a particular time in future
- Polling a device by checking its state at regular intervals when the hardware can’t fire interrupts.
- The user wants to send some messages to another device at regular intervals.
- Send an error when some action didn’t happen in a particular time period.

Kernel Timer API

- We need **#include <linux/timer.h>** in order to use kernel timers. Kernel timers are described by the timer_list structure, defined in **<linux/timer.h>**:

```
struct timer_list {
   /* ... */
   unsigned long expires;
   void (*function)(unsigned long);
   unsigned long data;
};
```

- The **expires** field contains the expiration time of the timer (in jiffies). On expiration, **function()** will be called with the given **data** value.

Initialize Kernel Timer

Init_timer

init_timer (struct timer_list * timer);

Timer_setup

Void timer_setup(timer, function, data);

Example

```
/* setup your timer to call my_timer_callback */
timer_setup(&hc_timer, timer_callback, 0);
//Timer Callback function. This will be called when timer expires
void timer_callback(struct timer_list  * data)
        {

        }
```

Initialize Kernel Timer cont.....

DEFINE_TIMER

```
DEFINE_TIMER(_name, _function, _expires, _data)
```

If we are using this method, then no need to create the `timer_list` structure on our side.

The kernel will create the structure in the name of `_name` and initialize it.

- `_name` – name of the timer_list structure to be created
- `_function` – Callback function to be called when the timer expires
- `_expires` – the expiration time of the timer (in jiffies)
- `_data` – data has to be given to the callback function

Start a kernel timers

add_timer

```
void add_timer(struct timer_list *timer);// this will start a timer
```

`timer` – the timer needs to be started

Modifying Kernel Timer’s timeout

Mod_timer

int mod_timer(struct timer_list * timer, unsigned long expires);

This function is used to modify a timer’s timeout. This is a more efficient way to update the **expires** field of an active timer (if the timer is inactive it will be activated).

mod_timer(timer, expires) is equivalent to:

```
del_timer(timer);
timer->expires = expires;
add_timer(timer);
```

timer – the timer needs to modify the timer period.

expires – the updated expiration time of the timer (in jiffies).

0 – **mod_timer** of an inactive timer
1 – **mod_timer** of an active timer

Stop a Kernel Timer

Del_timer

This will deactivate a timer. This works on both active and inactive timers

```
int del_timer(struct timer_list * timer);
```

Return value

0 – `del_timer` of an inactive timer
1 – `del_timer` of an active timer

Del_timer_sync

This will deactivate a timer and wait for the handler to finish. This works on both active and inactive timers.

```
int del_timer_sync(struct timer_list * timer);
```

Return value

0 – `del_timer_sync` of an inactive timer
1 – `del_timer_sync` of an active timer

Check Kernel Timer status

timer_pending

```
int timer_pending(const struct timer_list * timer);
```

Return value

0 – timer is not pending

1 – timer is pending

TASKLETS

- Tasklets are used to queue up work to be done at a later time.
- Tasklets are atomic, so we cannot use **`sleep()`** and such synchronization primitives as mutexes, semaphores, etc. from them. But we can use spinlock
- The major use of the tasklet is to schedule the bottom half of an interrupt service routine.

```
struct tasklet_struct {
        struct tasklet_struct *next;
        unsigned long state; //state is used to determine whether the tasklet has already been scheduled(i.e scheduled or running)
        atomic_t count; //It holds a nonzero value if the tasklet is disabled and 0 if it is enabled
        void (*func)(unsigned long); //func is a pointer to the function that will be run, with data as its parameter
        unsigned long data;
};
```

How to Create Tasklet

The below macros used to create a tasklet.

DECLARE_TASKLET

This macro used to create the tasklet structure and assigns the parameters to that structure.
If we are using this macro then the tasklet will be in the enabled state.

```
DECLARE_TASKLET(name, func, data);
```

`name` – name of the structure to be created.

`func` – This is the main function of the tasklet. Pointer to the function that needs to schedule for execution at a later time.

`data` – Data to be passed to the function "**`func`**".

Tasklet_schedule

Schedule a tasklet with a normal priority. If a tasklet has previously been scheduled (but not yet run), the new schedule will be silently discarded.

```
void tasklet_schedule (struct tasklet_struct *t);
```

t – pointer to the tasklet struct

Kill Tasklet

Finally, after a tasklet has been created, it’s possible to delete a tasklet through these below functions.
This will wait for its completion and then kill it.

```
void tasklet_kill( struct tasklet_struct *t );
```

t– pointer to the tasklet struct

Tasklet_kill_immediate

This is used only when a given CPU is in the dead state.

```
void tasklet_kill_immediate( struct tasklet_struct *t, unsigned int cpu );
```

t – pointer to the tasklet struct
cpu – CPU num

WORKQUEUES

- A workqueue contains a linked list of tasks to be run at a deferred time
  Tasks in workqueue
- run in process context, therefore can sleep, and without interfering with tasks running in any other queues.
- But still cannot transfer data to and from user space, as this is not a real user context to access

```
#include < linux/workqueue.h>

typedef void (*work_func_t)(struct work_struct *work);
struct work_struct {
	atomic_long_t data;
	struct list_head entry;
	work_funct_t func;
};
```

A work_struct can be declared and initialized at compile time with:

DECLARE_WORK(name, void (*function)(void *), void *data);

where name is the name of the structure which points to queueing up function() to run.

A previously declared work queue can be initialized and loaded with the the two macros: INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);

PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);

where work has already been declared as a work_struct.

The INIT_WORK() macro initializes the list_head linked-list pointer, and PREPARE_WORK() sets the function pointer.

The INIT_WORK() macro needs to be called at least once, and in turn calls PREPARE_WORK(). INIT_WORK() should not be called while a task is already in the

Alternatively, a workqueue can be statically declared by:

DECLARE_WORK(work, void (*function)(void *));

In the kernel, there is a default workqueue named events. Tasks are added to amd flushed from this queue with the functions:

```
int schedule_work(struct work_struct *work);
void flush_scheduled_work(void);
```

flush_scheduled_work() is used when one needs to wait until all entries in a work queue have run

Thank You