1 Timers (Optional)

1.1 jiffies

jiffies is a coarse time measurement variable provided by the linux kernel:

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

unsigned long volatile jiffies;
```

And here are some convenience macros compare two jiffies timestamps:

```
time_after(a, b);
time_before(a, b);
time_after_eq(a, b);
time_before_eq(a, b);
```

jiffies can be used to introduce busy waiting. For example:

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

jifdone = jiffies + delay * HZ;

while (time_before(jiffies, jifdone)); /* do nothing */
```

This kind of busy waiting is inefficient, as jiffies will be re-read every time it is accessed, and this loop locks up the CPU during the delay. Therefore busy waiting should not be used unless the wait time is very short (say under 50 jiffies).

Therefore for short delays, the following functions should be used instead:

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

void ndelay(unsigned long nanoseconds);
void udelay(unsigned long microseconds);
void mdelay(unsigned long milliseconds);
```

Don't expect **ndelay()** to give true nanoseconds. Instead, most architecture will give resolution up to microseconds.

Alteratively, these functions can be used, which do not have busy waiting:

```
void msleep (unsigned int milliseconds);
unsigned long msleep_interruptible (unsigned int milliseconds);
```

If **msleep_interruptible()** returns before the sleep has finished because of a signal, it returns the number of milliseconds left in the requested sleep period.

1.2 Timers

Timers are used to delay the execution of a function until a specified time has elapsed. The function will run on the CPU on which it is submitted.

Since the CPU may not be available when it is time to execute the function, therefore timers can only guarantee the function will not run before the specified time has elapsed. In practice, the function will run a clock tick after the timer expires, unless some greedy high latency tasks have been suspending interrupts.

Since the function scheduled will run in the interrupt (atomic) context instead of the process (user) context, therefore it cannot do anythings that cannot be done at interrupt time, including anything that can sleep (e.g. no transfer of data back and forth with user space, no semaphores, no memory allocation with GFP_KERNEL etc).

Here are the important data structure and functions for kernel timers:

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

struct timer_list {
    struct list_head entry;
    unsigned long expires;
    void (*function)(unsigned long);
    unsigned long data;
    struct tvec_t_base_s *base;
};

void init_timer     (struct timer_list *timer);
void add_timer      (struct timer_list *timer);
void mod_timer      (struct timer_list *timer, unsigned long expires);
void del_timer      (struct timer_list *timer);
void del_timer_sync (struct timer_list *timer);
```

where:

- **entry** points to the doubly-linked circular list of kernel timers.
- **expires** is measured in jiffies. It is an absolute value, not a relative one.
- The function to be run is passed as function() and data can be passed to it through the pointer argument data.
- **init_timer()** zeroes the previous and next pointers in the linked list.
- **add_timer()** inserts the timer into the global timer list.
- **mod_timer()** can be used to reset the time at which a timer expires.
- **del_timer()** can remove a timer before it expires. Returns **1** if it deletes the timer, or **0** if it is too late because the timer function has already started executing. It is not necessary to call **del_timer()** if the timer expires on its own.
- **del_timer_sync()** makes sure that upon return, the timer function is not running on any CPUs. This function should be used on SMP systems as it avoids race conditions.

A timer can reinstall itself to set up a periodic timer. This can be done by:

```
mod_timer(&t, jiffies + delay);
```

Here is a code snippet showing the usage of kernel timers:

```
static struct timer_list my_timer;

init_timer(&my_timer);

my_timer.function = my_function;
my_timer.expires = jiffies + ticks;
my_timer.data = &my_data;

add_timer(&my_timer);

.....

/* we don't need to execute my_function() anymore */
del_timer(&my_timer);
```

1.3 Exercise: Kernel Timers from a Character Driver

Write a driver that launches a kernel timer whenever a **write()** to the device takes place.

Pass some data to the driver and have it printed out.

Have it print out the **current→pid** field when the timer functions is scheduled, and then again when the function is executed.