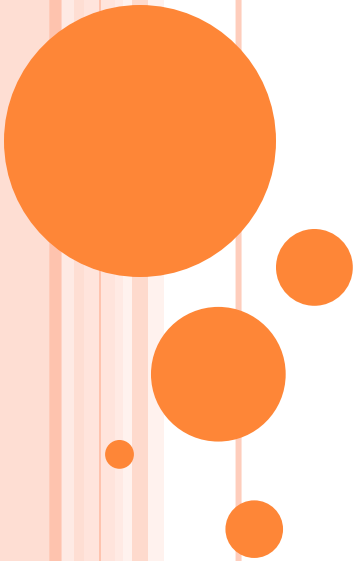

Linux Device Drivers Time, Delays and Deferred Work

CDAC Mumbai

HZ

- The kernel keeps track of the flow of time by means of timer interrupts. These are generated by the system's timing hardware at regular intervals.
- This interval is programmed at system boot up by the kernel according to the value HZ, which is an architecture dependent variable. The default values range from 50 to 1200 and is typically set to 100 or 1000 on x86 machines
- Changing the value of HZ to a new will take effect only on recompiling the kernel with the new value.

JIFFIES

- Every time a timer interrupt occurs, the value of an internal kernel counter is incremented. The counter is initialized to 0 on system boot and therefore represents the number of timer ticks since last boot.

- The counter is a 64 bit variable and is called jiffies_64.However, driver writers access the jiffies variable as an unsigned long that is same as either jiffies_64 or its least significant bits.

Using the Jiffies Counter

- The jiffies counter can be used in reading the present time and thereby calculating the future timestamp. This may be explained as follows
  - J = jiffies;
  - Stamp_1 = J + HZ

    //Stamp_1 iterates to one second ahead
  - Stamp_2 = J + HZ/2

    // Stamp_2 may refer to half a second in the future

Using the Jiffies Counter

- To compare the cached value with the current value, the following macros may be used
  - int time_after(unsigned long a, unsigned long b);
  - int time_before(unsigned long a, unsigned long b);
  - int time_after_eq(unsigned long a, unsigned long b);
  - int time_before_eq(unsigned long a, unsigned long b);

Delaying Execution

- Long Delays:
  - Occasionally a driver needs to delay execution for relatively long periods –more than one clock tick.
  - There are few ways of implementing the same
  - Busy Waiting

```
J = jiffies;
Delay = J + 5 * HZ //A delay of 5 HZ from now
While (time_before (J, Delay))
{
      /* do nothing */
}
```

Delaying Execution

- This method causes a busy looping in the while statement, which hogs the CPU for no productive outcome
- Yielding the Processor
  - While (time_before (J, Delay))

    {

    schedule(); // yield the CPU

    }
- The advantage of this method is that another process may get access to the CPU. The delay requested guaranteed but the process may not be scheduled exactly after the requested delay.

Delaying Execution

- Short Delays:
- The kernel implements functions that provide delays that may not otherwise be possible with the jiffies counter. These delays are implemented as function loop, depending on the architecture.

  - Void ndelay(unsigned long nsecs);
  - Void udelay(unsigned long usecs);
  - Void mdelay(unsigned long msecs;

Kernel Timers

- Kernel timers are used to schedule execution of a function at a later instance of time, based on the clock tick.
- A kernel timer is a data structure that instructs the kernel to execute a user defined function with a user defined argument at a user defined time.
- The declaration can be found in *linux/timer.h* and the source code may be found in *kernel/timer.c*

Kernel Timers

- The functions scheduled to run, may not run while the process that initiated it is executing. They are run asynchronously as in an interrupt context.
- Kernel timers may be considered as a software interrupt handlers and have certain constraints associated with their implementations.
- Primarily, they have to be atomic and there are additional constraints because of execution in the interrupt context
- The timers run on the same CPU that registered it.

Rules to be followed in Interrupt Context

- No user space access is allowed
- The Current Pointer is not meaningful in atomic mode
- No sleeping or scheduling may be performed

The Timer API

- The kernel provides drivers with a number of functions to declare, register and remove kernel timers

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

- The member *expires specifies the amount of* time for delay.
- There is a function pointer to a user defined function
- The third parameter data takes the arguments to the function pointer.

The Timer API

- The timer may be initialized by using the function
  - void init_timer(struct timer_list *timer);
- The public fields of the structure may be initialized after returning from the function
- This timer may be added and deleted to and from the kernel using the functions
  - void add_timer(struct timer_list *timer);
  - void del_timer(struct timer_list *timer);
- The timer is a one shot execution and is taken off the list before it is run.

Other Timer APIs

- int mod_timer(struct timer_list *timer, unsigned long expires);
- Allows the modification of the timer count

- int timer_pending(const struct timer_list *timer);
- Returns true or false to indicate whether the timer is currently scheduled to run by reading fields of the structure

Applications of Kernel Timers

- They find various applications such as polling a device by checking its status registers at regular intervals, when the hardware cannot fire interrupts.
- Other applications may be turning off the floppy motor, shutting down the processor fan on system shutdown etc...

Tasklets

- Tasklets is another kernel facility that allows deferring the execution of a process to a later instance.
- It is similar to kernel timers in that they run at interrupt time, they always run on the same CPU that schedules them and they receive an unsigned long argument
- They differ from kernel timers in the fact that they are not scheduled at a particular time. They are scheduled by the system at a later instance of time.

The Tasklet Data Structure

- A tasklet exists as a data structure that must be initialized before use.

```
struct tasklet_struct
        {
        void (* func)(unsigned long);
        unsigned long data;
        }
```

- Initialization is done by the function

```
void tasklet_init(    struct tasklet_struct t,
                      void(*func) (unsigned long),
                      unsigned long data          );
```

Tasklets

- A tasklet can be disabled and re-enabled later; it won't be executed until it is enabled as many times as it has been disabled
- A tasklet can re-register itself
- A tasklet can be scheduled to execute at normal priority or high priority
- Tasklets may be run immediately if the system is not under heavy load but never later than the next timer tick

Tasklet APIs

- void tasklet_disable(struct tasklet_struct *t);
- void tasklet_enable(struct tasklet_struct *t);
- void tasklet_schedule(struct tasklet_struct *t);
- void tasklet_hi_schedule(struct tasklet_struct *t);
- void tasklet_kill(struct tasklet_struct *t);

Work Queues

- Work queues allow the kernel code to request that a function be called at some future time. They differ as
  - Work Queue functions run in the context of a special kernel process
  - These functions can sleep
  - Kernel code can request that the execution of work queue functions be delayed for an explicit interval
- The key difference between tasklets and work queues is that tasklets execute for a short period, immediately and are atomic. The same does not hold for work queues

Work Queues

- A work queue must be explicitly created before use….
  - struct workqueue_struct *create_workqueue(const char *name);
- To submit a task to a work queue ,you need to fill in the work_struct structure(Static way, compile time)…..
  - DECLARE_WORK(name, void(*function)(void *) ,void *data);
- If you need to set up work_struct structure at runtime, use following macros…..
- INIT_WORK(struct work_struct *work, void(*function)(void *),void *data);
- PREPARE_WORK(struct work_struct *work, void(*function)(void*),void *data);
- There are two functions for submitting work to a workqueue
- Int queue_work(struct workqueue_struct *queue,struct work_struct *work);
- Int queue_delayed_work(struct workqueue_struct *queue,struct work_struct *work, unsigned long delay);

Tasklet vs Workqueue

TASKLET	WORKQUEUE
Tasklet execute quickly for a short period of time, and in atomic mode	functions may have higher latency, but need not be atomic
Tasklet runs in software interrupt context	Work queue functions run in the context of a special kernel process
Tasklet code must be atomic	Work queue function can sleep.