

# Dispatching Hardware Events in a Multi-Threaded Linux Application

By: Tony Khoshaba  
December 17, 2009

Proper dispatching of hardware interrupt events and other time critical events from Linux Kernel to a multi-threaded application is a challenging issue that requires delicate design consideration both at driver levels as well as application levels. With the programming model discussed in this article we have been able to design a successful robust system that would process a large number of interrupts and other timing events and produce desired reaction time in the system.

The basic idea is the use of wait queues at driver level and use of blocking threads on top of the driver for dispatching the events and the use of conditional signaling to propagate the events to other threads.

## **The Event Drivers**

The first component in such system is the interrupt or other timing sensitive drivers. Such driver will provide a blocking system read routine that would be unblocked by the interrupt event. To achieve this the following code is needed in the read routine:

```
int dev_read(struct file *file, char *buf, size_t count, loff_t *offset)
{
    ...
    interruptible_sleep_on(&queue);
    if(signal_pending(&current)) {
        return(-EINTR);
    }
    ...
}
```

In the interrupt or timer routines a call to

```
...
wake_up(&queue);
...
```

would be required to initiate the dispatching of the event. The dev\_read() routine would also update the necessary data to user memory space if needed.

## **The Event Dispatching Threads**

The event dispatching thread is a loop that blocks on a read() system call to the associated driver. Once

the interrupt event happens, `read()` unblocks and returns the needed data to application layer. Based on the content of the returned data, appropriate conditional signals are sent to all other threads which are waiting on a conditional signal that is serviced by this event dispatching thread. The event dispatching thread calls

```
...  
pthread_cond_signal(&cond_id);  
...
```

for different signals that go to different application threads.

## ***The Application Threads***

The application threads are loops that are conditional waiting on a signal to service an event. This is achieved by calling

```
...  
pthread_cond_timedwait(&cond_id, &mutex_id, &timeout);  
...
```

at the top of the loop. This routine will immediately unblocks upon reception of the corresponding conditional signal from event dispatching threads

## ***Other Considerations***

- The application thread need to use timed conditional wait to make use proper behavior and processing in the system even if the events do not happen.
- Use of global `mutex_id`'s between dispatching threads and application threads is required for better synchronization.

## ***Conclusion***

The programming model described here, if properly used and designed, is capable of handling multiple hardware interrupts and other kernel initiated timing events and processing the corresponding data by multiple thread with independent functionality and it is proven to be working robustly in a complex system such as a Monitor and Hot Standby (MHS) in microwave radio switches reaching a desirable almost real-time behavior.

For more information contact [tonykhosha@computingassociate.com](mailto:tonykhosha@computingassociate.com).