Wait Queue

Waitqueue is a list of processes all waiting for a specific event. It is made up of a spinlock and a list. To put a process to waitqueue you can call `wait_event_interruptible`. I have tried to cover the waitqueue implementation in kernel first and then i have discussed with an example, how it can be used in module/driver writing.

`wait_event_interruptible (wq, condition);`

`wq` - the waitqueue to wait on

`condition` - a C expression for the event to wait for

The process is put to sleep (TASK_INTERRUPTIBLE) until the condition evaluates to true or a signal is received. The condition is checked each time the waitqueue wq is woken up.

```
#define wait_event_interruptible(wq, condition)                   \
({                                                                \
    int __ret = 0;                                                \
    if (!(condition))                                             \
        __wait_event_interruptible(wq, condition, __ret);         \
        __ret;                                                    \
 })

#define __wait_event_interruptible(wq, condition, ret)            \
do {                                                              \
    DEFINE_WAIT(__wait);                                          \
    for (;;) {                                                    \
        prepare_to_wait(&wq, &__wait, TASK_INTERRUPTIBLE);        \
        if (condition)                                            \
            break;                                                \
        if (!signal_pending(current)) {                           \
            schedule();                                           \
            continue;                                             \
        }                                                         \
        ret = -ERESTARTSYS;                                       \
        break;                                                    \
     }                                                            \
     finish_wait(&wq, &__wait);                                   \
 } while (0)
```

The function will return -ERESTARTSYS if it was interrupted by a signal and 0 if condition evaluated to true.

The `wait_event_interruptible` intern calls `prepare_to_wait` -

```
prepare_to_wait(wait_queue_head_t *q, wait_queue_t *wait, int state)
 {
          unsigned long flags;

          wait->flags &= ~WQ_FLAG_EXCLUSIVE;
          spin_lock_irqsave(&q->lock, flags);
          if (list_empty(&wait->task_list))
                  __add_wait_queue(q, wait);
          set_current_state(state);
          spin_unlock_irqrestore(&q->lock, flags);
 }
 EXPORT_SYMBOL(prepare_to_wait);
```

The `prepare_to_wait` further calls `__add_wait_queue` which add the task to wait queue-

```
static inline void __add_wait_queue(wait_queue_head_t *head, wait_queue_t
*new)
 {
         list_add(&new->task_list, &head->task_list);
 }
```

Now, to wake up the task, some other process or interrupt handler has to perform wake up. The wake up is performed by calling `wake_up_interruptible`.

`wake_up_interruptible(wait_queue_head_t *queue);`

```
#define wake_up_interruptible(x)  __wake_up(x, TASK_INTERRUPTIBLE, 1, NULL)
```

The `wake_up_interruptible` is actually a macro for a function `__wake_up` -

```
void __wake_up(wait_queue_head_t *q, unsigned int mode, int nr_exclusive,
void *key)
 {
         unsigned long flags;
          spin_lock_irqsave(&q->lock, flags);
         __wake_up_common(q, mode, nr_exclusive, 0, key);
         spin_unlock_irqrestore(&q->lock, flags);
 }
 EXPORT_SYMBOL(__wake_up);
```

The `__wake_up` internally calls `__wake_up_common` which wakes up all the task in given wait queue -

```
static void __wake_up_common(wait_queue_head_t *q, unsigned int mode, int
nr_exclusive, int wake_flags, void *key)
 {
         wait_queue_t *curr, *next;
         list_for_each_entry_safe(curr, next, &q->task_list, task_list) {
                 unsigned flags = curr->flags;
                 if (curr->func(curr, mode, wake_flags, key) &&
                              (flags & WQ_FLAG_EXCLUSIVE) && !--nr_exclusive)
                         break;
         }
 }
```

Now, we have discussed waitqueue's implementation in the kernel. But, where and how to use the waitqueue? Lets discuss this in following example.

Example -

Suppose you want to read from a device. The device sends data at random times. So, in device driver you can write an interrupt handler which reads from the device and stores it in some internal buffer. The interrupt handler will only execute when it receives an interrupt from the device.

To read from the device, you have provided a read call. When the user made a call to read the device, it is not necessary that the device provide the data immediately. There are two possibilities. The process who made the read call may block until the data is available or it the read call may fail saying that data is not available.

When a read call is made to block until the data is available, it is implemented using waitqueues. You wait in read method of you module using `wait_event_interruptible` until the flag becomes 1, which will only happen in interrupt handler. When an interrupt arrives, the interrupt handler of the module is executed. In the interrupt handler all necessary handling is performed and data received from the device is stored in some buffer. After all the copying of data and all handling is done, the task is woken up using `wake_up_interruptible`. As soon as a call to `wake_up_interruptible` is made, the blocked task is taken out of the wait queue and the condition is checked which is now found to be true and the read procedure continues from the point where it was blocked earlier. It must be noted that flag should be changed i.e. the blocking condition should be made true before calling `wake_up_interruptible`.You also need to do some initialisation of the waitqueue before you use it. It is also shown in the sample code below, how you can initialize the waitqueues and how you can use it.

```
wait_queue_head_t my_wq;
init_waitqueue_head(&may_wq);

int flag = 0;

interrupt_handler(...) {
...
...
  flag = 1;
  wake_up_interruptible(&my_wq);
}

read(...) {
...
...
  wait_event_interruptible(my_wq, flag!=0);
  flag = 0;
...
...
}
```

Well, i said earlier that a read call can be blocking or non-blocking. But, how your module/driver knows that you want blocking or non-blocking call. This can be determined by a flag you pass with the read system call.

You can pass the flag as `O_NONBLOCK` or `O_NDELAY` if you want the read to be non-blocking. If you don't specify any flag in the call you made, it is taken as blocking. This is because by default the behavior is blocking.

In the module/driver you can check this flag from `filp->f_flag` which contains this flag. So, if the non-blocking call is made don't use `wait_event_interruptible`.

Happy coding!!.