LINUX CHARACTER DEVICE DRIVERS II

Important Notice : Courseware - Legal

2000-2020 Kaiwan N Billimoria
kaiwanTECH, Bangalore, India.

kaiwanTECH Linux OS Corporate Training Programs
Please do check out our current offering of world-class, seriously-valuable, high on returns, technical Linux OS corporate training programs here: http://bit.ly/ktcorp

Blocking I/O [1]

One problem that might arise with read is what to do when there's no data yet, but we're not at end-of-file.

The default answer is **"go to sleep waiting for data".** This section shows how a process is put to sleep, how it is awakened, and how an application can ask if there is data without just blindly issuing a read call and blocking.

Going to Sleep and Awakening

Whenever a process must wait for an event (such as the arrival of data or the termination of a process), it should go to sleep. Sleeping causes the process to suspend execution, freeing the processor for other uses. At some future time, when the event being waited for occurs, the process will be woken up and will continue with its job. This section discusses the kernel machinery for putting a process to sleep and waking it up.

Wait Queues are a kernel mechanism for allowing tasks to wait upon (sleep on) an event and be awoken when it occurs.

There are several ways of handling sleeping and waking up in Linux, each suited to different needs. All, however, work with the same basic data type, a **wait queue** (wait_queue_head_t). A wait queue is exactly that - a linked list of processes that are waiting for an event.
Wait queues are declared and initialized as follows:

```
wait_queue_head_t my_queue;
init_waitqueue_head(&my_queue);
```

When a wait queue is declared statically (i.e., not as an automatic variable of a procedure or as part of a dynamically-allocated data structure), it is also possible to initialize the queue at compile time:

```
DECLARE_WAIT_QUEUE_HEAD(my_queue);
```

It is a common mistake to neglect to initialize a wait queue (especially since earlier versions of the kernel did not require this initialization); if you forget, the results will usually not be what you intended.

When a process sleeps, it does so in expectation that some condition will become true in the future. Any process that sleeps must check to be sure that the condition it was waiting

[1] This discussion is extracted from "Linux Device Drivers" 3rd Edition by A Rubini, J Corbet & GK-Hartman

for is really true when it wakes up again.

The simplest way of sleeping in the Linux kernel is a macro called wait_event (with a few variants); it combines handling the details of sleeping with a check on the condition a process is waiting for. The forms of wait_event are:

```
wait_event(wait_queue_head_t queue, int condition);
wait_event_interruptible(wait_queue_head_t queue,
            int condition);

wait_event_timeout(wait_queue_head_t queue,
            int condition, long timeout);
wait_event_interruptible_timeout(
      wait_queue_head_t queue, int condition,
      long timeout);
```

The *wait_event_timeout* and *wait_event_interruptible_timeout* wait for a limited time; after that time period (**expressed in jiffies**) expires, the macros return with a value of 0 regardless of how *condition* evaluates. How does one convert the time interval from “n” seconds to jiffies?
Simple:

```
(the jiffies equivalent of) timeout = jiffies + (n*HZ)
```

SIDEBAR :: xxx_to_jiffies Kernel API

Other convenience APIs:
 	
`unsigned long msecs_to_jiffies(const unsigned int m);`
`unsigned long usecs_to_jiffies(const unsigned int u);`
`unsigned long nsecs_to_jiffies(u64 n);`
`unsigned long timespec_to_jiffies(const struct timespec *value);`
`unsigned long timeval_to_jiffies(const struct timeval *value) ;`
`unsigned long clock_t_to_jiffies(unsigned long x);`

In all of the above forms, queue is the wait queue head to use. Notice that it is passed “by value.” The *condition is an arbitrary boolean C expression* that is evaluated by the macro before and after sleeping; until condition evaluates to a true value, the process continues to sleep. Note that condition may be evaluated an arbitrary number of times, so it should not have any side effects.

If you use *wait_event* , your process is put into an **uninterruptible sleep** which, as we have mentioned before, is usually not what you want. The preferred alternative is *wait_event_interruptible* , which **can be interrupted by signals**. This version returns an integer value that you should check; a nonzero value means your *sleep was interrupted by some sort of signal*, and your driver should probably return -ERESTARTSYS .

SIDEBAR | What exactly does the return value -ERESTARTSYS mean?

This question was asked on stackoverflow; the detailed answer is reproduced below:

-ERESTARTSYS is connected to the concept of a restartable system call. A restartable system call is one that can be transparently re-executed by the kernel when there is some interruption.

For instance the user space process which is sleeping in a system call can get a signal, execute a handler, and then when the handler returns, it appears to go back into the kernel and keeps sleeping on the original system call.

Using the sigaction API, processes can arrange the restart behavior associated with signals. This is all part of POSIX. Look up sigaction.

In the Linux kernel, when a driver or other module blocking in the context of a system call detects that a task has been woken because of a signal, it can return -EINTR. But -EINTR will bubble up to user space and cause the system call to return -1 with errno set to EINTR.

If you return -ERESTARTSYS instead, it means that your system call is restartable. The ERESTARTSYS code will not be seen in user space. It either gets translated to a -1 return and errno set to EINTR, or it is translated into a system call restart behavior, which means that your syscall is called again with the same arguments (by no action on part of the user space process: the kernel does this by stashing the info in a special restart block).

Note the obvious problem: some system calls can't be restarted with the same parameters! For instance, suppose there is a sleep call like nanosleep, for 5.3 seconds. It gets interrupted after 5 seconds. If it restarts naively, it will sleep for another 5.3 seconds. It has to pass new parameters to the restarted call to sleep for only the remaining 0.3 seconds; i.e. alter the contents of the restart block. There is a way to do that: you stuff different arguments into the task's restart block and use the -ERESTART_RESTARTBLOCK return value.

To address the second question: what's the difference? Why not just write the read routine without checking the return value and returning -ERESTARTSYS? Well, because that is incorrect in the case that the wakeup is due to a signal! Do you want a read to return 0 bytes read whenever a signal arrives? That could be misinterpreted by user space as end of data. This kind of problem won't show up in test cases that don't use signals.

share|edit|flag
edited **Mar 6 at 1:57**
answered **Mar 6 at 1:49**
Kaz
7,168316

<< FYI: 2.6.25 – new feature; see http://kernelnewbies.org/LinuxChanges >>

Most Unix systems have two states when sleeping -- interruptible and uninterruptible. 2.6.25 adds a third state: **killable**. While interruptible sleeps can be interrupted by any signal, killable sleeps can only be interrupted by fatal signals. The practical implications of this feature is that NFS has been converted to use it, and as a result you can now kill -9 a task that is waiting for an NFS server that isn't contactable. Further uses include allowing the OOM killer to make better decisions (it can't kill a task that's sleeping uninterruptibly) and changing more parts of the kernel to use the killable state. ...

To use this feature, call the macro

```
      wait_event_killable(wait_queue_head_t queue,
                          int condition);
```

```
Also:
wait_event_freezable(wq_head, condition);
wait_event_freezable_timeout(wq_head, condition, timeout);
>>
```

<<

There are variations on the wait_event; a quick grep reveals all:

```
$ grep "^#define.*[^_]wait_" include/linux/wait.h
#define wait_event(wq_head, condition)                                    \
#define wait_event_freezable(wq_head, condition)                          \
#define wait_event_timeout(wq_head, condition, timeout)                   \
#define wait_event_freezable_timeout(wq_head, condition, timeout)         \
#define wait_event_exclusive_cmd(wq_head, condition, cmd1, cmd2)          \
#define wait_event_cmd(wq_head, condition, cmd1, cmd2)                    \
#define wait_event_interruptible(wq_head, condition)                      \
#define wait_event_interruptible_timeout(wq_head, condition, timeout)     \
#define wait_event_hrtimeout(wq_head, condition, timeout)                 \
#define wait_event_interruptible_hrtimeout(wq, condition, timeout)        \
#define wait_event_interruptible_exclusive(wq, condition)                 \
#define wait_event_killable_exclusive(wq, condition)                      \
#define wait_event_freezable_exclusive(wq, condition)                     \
#define wait_event_idle(wq_head, condition)                               \
#define wait_event_idle_exclusive(wq_head, condition)                     \
#define wait_event_idle_timeout(wq_head, condition, timeout)              \
#define wait_event_idle_exclusive_timeout(wq_head, condition, timeout)    \
#define wait_event_interruptible_locked(wq, condition)                    \
#define wait_event_interruptible_locked_irq(wq, condition)                \
#define wait_event_interruptible_exclusive_locked(wq, condition)          \
#define wait_event_interruptible_exclusive_locked_irq(wq, condition)      \
#define wait_event_killable(wq_head, condition)                           \
#define wait_event_killable_timeout(wq_head, condition, timeout)          \
#define wait_event_lock_irq_cmd(wq_head, condition, lock, cmd)            \
#define wait_event_lock_irq(wq_head, condition, lock)                     \
#define wait_event_interruptible_lock_irq_cmd(wq_head, condition, lock, cmd)
       \
#define wait_event_interruptible_lock_irq(wq_head, condition, lock)       \
#define wait_event_interruptible_lock_irq_timeout(wq_head, condition, lock,\
#define wait_event_lock_irq_timeout(wq_head, condition, lock, timeout)    \
$
>>
```

Awakening from a Wait Queue

Of course, sleeping is only half of the solution; something, somewhere ***will have to wake the process up again***. When a device driver sleeps directly, there is usually code in another part of the driver that performs the wakeup, once it knows that the event has occurred. Typically a driver **will wake up sleepers in its interrupt handler once new data has arrived**. Other scenarios are possible, however.

Just as there is more than one way to sleep, so there is also more than one way to wake up. The high-level functions provided by the kernel to wake up processes are as follows:

wake_up(wait_queue_head_t *queue);

This function will wake up *all processes that are waiting* on this event queue.

Actually, that's not the full picture. *wake_up* awakens every process on the queue that is not in an exclusive wait, and exactly one exclusive waiter, if it exists (there is a special flag to identify an exclusive waiter- WQ_FLAG_EXCLUSIVE; to use this, setting up an exclusive wait, use the `wait_event_interruptible_exclusive` macro; or could use the prepare_to_wait_exclusive().You could see more details in the LDD3 book.)

wake_up_interruptible does the same, with the exception that it skips over processes in an uninterruptible sleep. These functions can, before returning, cause one or more of the processes awakened to be scheduled (although this does not happen if they are called from an atomic context).

wake_up_interruptible(wait_queue_head_t *queue);

If your driver is using wait_event_interruptible, there is little difference between wake_up and wake_up_interruptible. Calling the latter is a common convention, however, to preserve consistency between the two calls.

When the *wake_up** call is issued, *all* tasks that were asleep are awakened. However, the driver can respond in one of two ways; it can have:
a) All 'n' tasks *stay awake and perform the operation* (read, write or whatever they were

waiting to do)
b) Only one task remains awake, all the other 'n-1' tasks go to sleep awaiting their turn, and it performs the required operation(s).

How is this achieved? Easy – behaviour (a) is the default. If you want behaviour (b), use a concurrency control mechanism (typically a semaphore / mutex – we seen how exactly to use a mutex in the next section 'Writing Reentrant Code').

Thus this is entirely in the control of the driver author (and is demonstrated below in the "sleepy" driver example slpy.c).

Additional calls (*wake_up_[interruptible_]all*) are available to wake up *all* tasks on a wait queue, if so desired.

As an example of wait queue usage, imagine you want to put a process to sleep when it reads your device and awaken it when someone else writes to the device. The following sample driver does just that.

Participants can find the source code of the "sleepy" driver on the GitHub repo specified by the instructor.

We can see that putting a task to sleep is easy for the driver to do. However, a few **important rules** to keep in mind:

1. Never sleep when running in an atomic context
2. Don't make assumptions as to the state of the system when you are awoken; almost anything could have changed while you were asleep. Recheck that the condition that you were waiting for has actually arisen. *Any code that sleeps should do so in a loop that tests the condition after returning from the sleep*. *<< see the next section >>*
3. You must think through your code, ensuring that every sleep has a corresponding *wakeup* somewhere in the driver
4. It's in the driver author's hands to control whether a wakeup should be treated as to wake up *and* put into the running state all previously sleeping tasks, or just one of them – by using a concurrency control mechanism, such as a mutex. Thus, depending on the specifics of the driver, you decide which semantic is appropriate – do you intend a "destructive" or "non-destructive" read.
5. If we take the case of *all* tasks (or one) being awoken, then another question arises: *which* one of the tasks will actually run first?
   The answer is that it's indeterminate and a robust application should not make any assumption regarding the order in which sleeping tasks are awakened – it will appear more-or-less random. *<< 2.6.25 onwards - "ordered wakeup" mechanism ??>>*

Example usage of wait queues:

- can be seen in the RTC driver (*drivers/char/rtc.c*)
- this GitHub gist.

Writing Reentrant Code

When a process is put to sleep, the driver is still alive and can be called by another process.

Of course, **on SMP** systems, **multiple simultaneous calls** to your driver can happen even when you do not sleep.

Such situations can be **handled painlessly by writing reentrant code**. Reentrant code is code that **doesn't keep status information in global variables** and thus is able to manage interwoven invocations without mixing anything up. If all the status information is **process specific**, no interference will ever happen.

If status information is needed, it can either **be kept in local variables** within the driver function (each process – each *thread* actually - has a different stack page in kernel space where local variables are stored), or it can reside in **private_data** within the filp accessing the file. Using local variables is preferred because sometimes the same filp can be shared between two processes (usually parent and child) << and multiple threads, and as a side effect of the dup / dup2 system calls. >>

If you need to save large amounts of status data, you can keep the pointer in a local variable and use kmalloc to retrieve the actual storage space. In this case you must remember to kfree the data, because there's no equivalent to "everything is released at process termination" when you're working in kernel space. Using local variables for large items is not good practice, because the data may not fit the **single page of memory allocated for stack space (for each process context in kernel space)**.
...

Finally, of course, code that sleeps should always keep in mind that the state of the system **can change** in almost any way while a process is sleeping. The driver should be careful to check any aspect of its environment that might have changed while it wasn't paying attention.

An Example of Handling Blocking / Non-blocking I/O and Reentrant-Safe Driver Coding

The psuedo-code below demonstrates typical code that you would use for a read implementation manages *both* blocking and nonblocking input (we do not take into account the usage of *global variables* here; lets look at that aspect of concurrency control later):

```
...
ssize_t dev_read (struct file *filp, char __user *buf, size_t count,
                  loff_t *f_pos)
{
        Dev * dev = (Dev *)filp->private_data;

        /* lock the mutex */
        if (mutex_lock_interruptible(&dev->mutex_lock))
                return -ERESTARTSYS;

        while ( <condition> ) { /* condition : if true => nothing to read */
                mutex_unlock(&dev->mutex_lock); /* release the lock */

                if (filp->f_flags & O_NONBLOCK) /* non-blocking I/O, try again */
                        return -EAGAIN;

                printk( KERN_INFO "\"%s\" reading: going to sleep\n", current->comm);
                if( wait_event_interruptible( waitq_read,
                                <condition_to_wait_for> )
                        return -ERESTARTSYS; /* signal: tell the fs layer to handle it */

                /* We're (all) out of the sleep, first check if data has really become
        available; otherwise loop, but first reacquire the lock - because of the "thundering
        herd" ... */
                if (mutex_lock_interruptible(&dev->mutex_lock))
                        return -ERESTARTSYS;
        } // while

        /* Ok, data is there, return something.
         * Do the hardware specific stuff to get the data into your
         * kernel buffer dev->from.
         * << ... >>
         */
        if (copy_to_user(buf, dev->from, count)) {
                mutex_unlock (&dev->mutex_lock);
                return -EFAULT;
        }

        mutex_unlock  (&dev->mutex_lock);       /* unlock */

        /* finally, awaken any writers (if there is an output buffer and
         * a wait queue to use it) and return */
        wake_up_interruptible(&dev->outq);
        return count;
}
```

Note also, once again, the use of **semaphores (now mutexes)** to protect critical regions of the code. The code has to be careful *to avoid going to sleep when it holds a mutex* -- otherwise, writers would never be able to add data, and the whole thing would **deadlock**.

This code uses *wait_event_interruptible* to wait for data if need be; it has to check for available data again after the wait, though. Somebody else could grab the data between when we wake up and when we get the semaphore back.

It's worth repeating that a process *can go to sleep* both when it calls schedule(), either directly or indirectly, and when it copies data to or from user space.

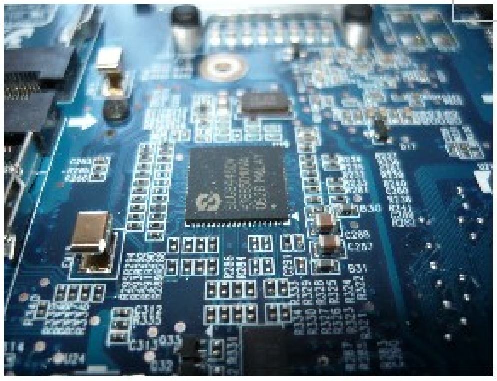
In the latter case the process may sleep if the user array is *not currently present in main memory*. If the driver causes the process to sleep while copying data between kernel and user space, it will sleep with the device semaphore held. Holding the semaphore in this case is justified since it will not deadlock the system, and since it is important that the device memory array not change while the driver sleeps.

The if statement that follows *wait_event_interruptible* takes care of signal handling. This statement ensures the proper and expected reaction to signals, which could have been responsible for waking up the process (since we were in an interruptible sleep).

If a signal has *arrived and it has not been blocked* by the process, the proper behavior is to let upper layers of the kernel handle the event. To this aim, the *driver returns* -ERESTARTSYS to the caller; this value is used internally by the virtual filesystem (VFS) layer, which *either restarts the system call or returns* -EINTR *to user space*.

Linux Operating System Specialized

kaiwanTECH



The highest quality Training on:

Linux Fundamentals, CLI and Scripting
Linux Systems Programming
Linux Kernel Internals
Linux Device Drivers
Embedded Linux
Linux Debugging Techniques
New! *Linux OS for Technical Managers*

Please do visit our website for details:
http://kaiwantech.in