Kernel Korner

Sleeping in the Kernel

Kedar Sovani

Abstract

The old sleep_on() function won't work reliably in an age of SMP systems and hyperthreaded processors. Here's how to make a process sleep in a safe, cross-platform way.

In Linux kernel programming, there are numerous occasions when processes wait until something occurs or when sleeping processes need to be woken up to get some work done. There are different ways to achieve these things.

All of the discussion in this article refers to kernel mode execution. A reference to a process means execution in kernel space in the context of that process.

Some kernel code examples have been reformatted to fit this print format. Line numbers refer to lines in the original file.

The schedule() Function

In Linux, the ready-to-run processes are maintained on a run queue. A ready-to-run process has the state TASK_RUNNING. Once the timeslice of a running process is over, the Linux scheduler picks up another appropriate process from the run queue and allocates CPU power to that process.

A process also voluntarily can relinquish the CPU. The schedule() function could be used by a process to indicate voluntarily to the scheduler that it can schedule some other process on the processor.

Once the process is scheduled back again, execution begins from the point where the process had stopped—that is, execution begins from the call to the schedule() function.

At times, processes want to wait until a certain event occurs, such as a device to initialise, I/O to complete or a timer to expire. In such a case, the process is said to sleep on that event. A process can go to sleep using the schedule() function. The following code puts the executing process to sleep:

```c
sleeping_task = current;
set_current_state(TASK_INTERRUPTIBLE);
schedule();
func1();
/* The rest of the code */
```

Now, let's take a look at what is happening in there. In the first statement, we store a reference to this process' task structure. current, which really is a macro, gives a pointer to the executing process' task_structure.

set_current_state changes the state of the currently executing process from TASK_RUNNING to TASK_INTERRUPTIBLE. In this case, as mentioned above, the schedule() function simply should schedule another process. But that happens only if the state of the task is TASK_RUNNING. When the schedule() function is called with the state as TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE, an additional
step is performed: the currently executing process is moved off the run queue before another process is scheduled. The effect of this is the executing process goes to sleep, as it no longer is on the run queue. Hence, it never is scheduled by the scheduler. And, that is how a process can sleep.

Now let's wake it up. Given a reference to a task structure, the process could be woken up by calling:

```
wake_up_process(sleeping_task);
```

As you might have guessed, this sets the task state to TASK_RUNNING and puts the task back on the run queue. Of course, the process runs only when the scheduler looks at it the next time around.

So now you know the simplest way of sleeping and waking in the kernel.

**Interruptible and Uninterruptible Sleep**

A process can sleep in two different modes, interruptible and uninterruptible. In an interruptible sleep, the process could be woken up for processing of signals. In an uninterruptible sleep, the process could not be woken up other than by issuing an explicit wake_up. Interruptible sleep is the preferred way of sleeping, unless there is a situation in which signals cannot be handled at all, such as device I/O.

**Lost Wake-Up Problem**

Almost always, processes go to sleep after checking some condition. The lost wake-up problem arises out of a race condition that occurs while a process goes to conditional sleep. It is a classic problem in operating systems.

Consider two processes, A and B. Process A is processing from a list, consumer, while the process B is adding to this list, producer. When the list is empty, process A sleeps. Process B wakes A up when it appends anything to the list. The code looks like this:

```
Process A:
1  spin_lock(&list_lock);
2  if(list_empty(&list_head)) {
3      spin_unlock(&list_lock);
4      set_current_state(TASK_INTERRUPTIBLE);
5      schedule();
6      spin_lock(&list_lock);
7  }
8
9  /* Rest of the code ... */
10  spin_unlock(&list_lock);

Process B:
100 spin_lock(&list_lock);
101 list_add_tail(&list_head, new_node);
102 spin_unlock(&list_lock);
103 wake_up_process(processa_task);
```

There is one problem with this situation. It may happen that after process A executes line 3 but before it executes line 4, process B is scheduled on another processor. In this timeslice, process B executes all its instructions, 100 through 103. Thus, it performs a wake-up on process A, which has not yet gone to sleep. Now, process A, wrongly assuming that it safely has performed the check for list_empty, sets the state to TASK_INTERRUPTIBLE and goes to sleep.
Thus, a wake up from process B is lost. This is known as the lost wake-up problem. Process A sleeps, even though there are nodes available on the list.

This problem could be avoided by restructuring the code for process A in the following manner:

```c
1  set_current_state(TASK_INTERRUPTIBLE);
2  spin_lock(&list_lock);
3  if(list_empty(&list_head)) {
4    spin_unlock(&list_lock);
5    schedule();
6    spin_lock(&list_lock);
7  }
8  set_current_state(TASK_RUNNING);
9
10 /* Rest of the code ... */
11 spin_unlock(&list_lock);
```

This code avoids the lost wake-up problem. How? We have changed our current state to TASK_INTERRUPTIBLE, before we test the condition. So, what has changed? The change is that whenever a `wake_up_process` is called for a process whose state is TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE, and the process has not yet called `schedule()`, the state of the process is changed back to TASK_RUNNING.

Thus, in the above example, even if a wake-up is delivered by process B at any point after the check for `list_empty` is made, the state of A automatically is changed to TASK_RUNNING. Hence, the call to `schedule()` does not put process A to sleep; it merely schedules it out for a while, as discussed earlier. Thus, the wake-up no longer is lost.

Here is a code snippet of a real-life example from the Linux kernel (linux-2.6.11/kernel/sched.c: 4254):

```c
4253 /* Wait for kthread_stop */
4254 set_current_state(TASK_INTERRUPTIBLE);
4255 while (!kthread_should_stop()) {
4256   schedule();
4257  set_current_state(TASK_INTERRUPTIBLE);
4258 }
4259 __set_current_state(TASK_RUNNING);
4260 return 0;
```

This code belongs to the migration_thread. The thread cannot exit until the kthread_should_stop() function returns 1. The thread sleeps while waiting for the function to return 0.

As can be seen from the code, the check for the kthread_should_stop condition is made only after the state is TASK_INTERRUPTIBLE. Hence, the wake-up received after the condition check but before the call to `schedule()` function is not lost.

## Wait Queues

Wait queues are a higher-level mechanism used to put processes to sleep and wake them up. In most instances, you use wait queues. They are needed when more than one process wants to sleep on the occurrence of one or more than one event.

A wait queue for an event is a list of nodes. Each node points to a process waiting for that event. An
individual node in this list is called a wait queue entry. Processes that want to sleep while the event occurs add themselves to this list before going to sleep. On the occurrence of the event, one or more processes on the list are woken up. Upon waking up, the processes remove themselves from the list.

A wait queue could be defined and initialised in the following manner:

```c
wait_queue_head_t my_event;
init_waitqueue_head(&my_event);
```

The same effect could be achieved by using this macro:

```c
DECLARE_WAIT_QUEUE_HEAD(my_event);
```

Any process that wants to wait on my_event could use either of the following options:

1. `wait_event(&my_event, (event_present == 1) );`
2. `wait_event_interruptible(&my_event, (event_present == 1) );`

The interruptible version 2 of the options above puts the process to an interruptible sleep, whereas the other (option 1) puts the process into an uninterruptible sleep.

In most instances, a process goes to sleep only after checking some condition for the availability of the resource. To facilitate that, both these functions take an expression as the second argument. The process goes to sleep only if the expression evaluates to false. Care is taken to avoid the lost wake-up problem.

Old kernel versions used the functions `sleep_on()` and `interruptible_sleep_on()`, but those two functions can introduce bad race conditions and should not be used.

Let's now take a look at some of the calls for waking up process sleeping on a wait queue:

1. `wake_up(&my_event);`: wakes up only one process from the wait queue.
2. `wake_up_all(&my_event);`: wakes up all the processes on the wait queue.
3. `wake_up_interruptible(&my_event);`: wakes up only one process from the wait queue that is in interruptible sleep.

**Wait Queues: Putting It Together**

Let us look at a real-life example of how wait queues are used. smbiod is the I/O thread that performs I/O operations for the SMB filesystem. Here is a code snippet for the smbiod thread (linux-2.6.11/fs/smbfs/smbiod.c: 291):

```c
static int smbiod(void *unused)
{
    daemonize("smbiod");
    allow_signal(SIGKILL);
    VERBOSE("SMB Kernel thread starting " 
        "(%d)...\n", current->pid);
    for (;;) {
```
As is clear from the code, smbiod is a thread that runs in a continuous loop as it processes I/O requests. When there are no I/O requests to process, the thread goes to sleep on the wait queue smbiod_wait. This is achieved by calling `wait_event_interruptible` (line 304). This call causes the smbiod to sleep only if the DATA_READY bit is set. As mentioned earlier, `wait_event_interruptible` takes care to avoid the lost wake-up problem.

Now, when a process wants to get some I/O done, it sets the DATA_READY bit in the smbiod_flags and wakes up the smbiod thread to perform I/O. This can be seen in the following code snippet (linux-2.6.11/fs/smbfs/smbiod.c: 57):

```c
void smbiod_wake_up(void)
{
    if (smbiod_state == SMBIOD_DEAD)
        return;
    set_bit(SMBIOD_DATA_READY, &smbiod_flags);
    wake_up_interruptible(&smbiod_wait);
}
```

`wake_up_interruptible` wakes up one process that was sleeping on the smbiod_wait waitqueue. The function `smb_add_request` (linux-2.6.11/fs/smbfs/request.c: 279) calls the `smbiod_wake_up` function when it adds new requests for processing.

**Thundering Herd Problem**

Another classical operating system problem arises due to the use of the `wake_up_all` function. Let us consider a scenario in which a set of processes are sleeping on a wait queue, wanting to acquire a lock.

Once the process that has acquired the lock is done with it, it releases the lock and wakes up all the processes sleeping on the wait queue. All the processes try to grab the lock. Eventually, only one of these acquires the lock and the rest go back to sleep.

This behavior is not good for performance. If we already know that only one process is going to resume
while the rest of the processes go back to sleep again, why wake them up in the first place? It consumes valuable CPU cycles and incurs context-switching overheads. This problem is called the thundering herd problem. That is why using the wake_up_all function should be done carefully, only when you know that it is required. Otherwise, go ahead and use the wake_up function that wakes up only one process at a time.

So, when would the wake_up_all function be used? It is used in scenarios when processes want to take a shared lock on something. For example, processes waiting to read data on a page could all be woken up at the same moment.

**Time-Bound Sleep**

You frequently may want to delay the execution of your process for a given amount of time. It may be required to allow the hardware to catch up or to carry out an activity after specified time intervals, such as polling a device, flushing data to disk or retransmitting a network request. This can be achieved by the function schedule_timeout(timeout), a variant of schedule(). This function puts the process to sleep until timeout jiffies have elapsed. jiffies is a kernel variable that is incremented for every timer interrupt.

As with schedule(), the state of the process has to be changed to TASK_INTERRUPTIBLE/TASK_UNINTERRUPTIBLE before calling this function. If the process is woken up earlier than timeout jiffies have elapsed, the number of jiffies left is returned; otherwise, zero is returned.

Let us take a look at a real-life example (linux-2.6.11/arch/i386/kernel/apm.c: 1415):

```c
1415  set_current_state(TASK_INTERRUPTIBLE);
1416  for (;;) {
1417     schedule_timeout(APM_CHECK_TIMEOUT);
1418     if (exit_kapmd)
1419         break;
1421     * Ok, check all events, check for idle
1422     .... * (and mark us sleeping so as not to
1423     .... * count towards the load average)..
1424     */
1424      set_current_state(TASK_INTERRUPTIBLE);
1425      apm_event_handler();
1426  }
```

This code belongs to the APM thread. The thread polls the APM BIOS for events at intervals of APM_CHECK_TIMEOUT jiffies. As can be seen from the code, the thread calls schedule_timeout() to sleep for the given duration of time, after which it calls apm_event_handler() to process any events.

You also may use a more convenient API, with which you can specify time in milliseconds and seconds:

1. msleep(time_in_msec);
2. msleep_interruptible(time_in_msec);
3. ssleep(time_in_sec);

msleep(time_in_msec); and msleep_interruptible(time_in_msec); accept the time to sleep in milliseconds, while ssleep(time_in_sec); accepts the time to sleep in seconds. These higher-level routines internally convert the time into jiffies, appropriately change the state of the process and call schedule_timeout(), thus making the process sleep.

I hope that you now have a basic understanding of how processes safely can sleep and wake up in the kernel.
To understand the internal working of wait queues and advanced uses, look at the implementations of `init_waitqueue_head`, as well as variants of `wait_event` and `wake_up`.

**Acknowledgement**

Greg Kroah-Hartman reviewed a draft of this article and contributed valuable suggestions.