Completions - "wait for completion" barrier APIs

Introduction:

If you have one or more threads that must wait for some kernel activity to have reached a point or a specific state, completions can provide a race-free solution to this problem. Semantically they are somewhat like a pthread_barrier() and have similar use-cases.

Completions are a code synchronization mechanism which is preferable to any misuse of locks/semaphores and busy-loops. Any time you think of using yield() or some quirky msleep(1) loop to allow something else to proceed, you probably want to look into using one of the wait_for_completion*() calls and complete() instead.

The advantage of using completions is that they have a well defined, focused purpose which makes it very easy to see the intent of the code, but they also result in more efficient code as all threads can continue execution until the result is actually needed, and both the waiting and the signalling is highly efficient using low level scheduler sleep/wakeup facilities.

Completions are built on top of the waitqueue and wakeup infrastructure of the Linux scheduler. The event the threads on the waitqueue are waiting for is reduced to a simple flag in 'struct completion', appropriately called "done".

As completions are scheduling related, the code can be found in kernel/sched/completion.c.

Usage:

There are three main parts to using completions:

- the initialization of the 'struct completion' synchronization object
- the waiting part through a call to one of the variants of wait_for_completion(),
- the signaling side through a call to complete() or complete_all().

There are also some helper functions for checking the state of completions. Note that while initialization must happen first, the waiting and signaling part can happen in any order. I.e. it's entirely normal for a thread to have marked a completion as 'done' before another thread checks whether it has to wait for it.

To use completions you need to #include <linux/completion.h> and create a static or dynamic variable of type 'struct completion', which has only two fields:

```
struct completion {
        unsigned int done;
        wait_queue_head_t wait;
};
```

This provides the ->wait waitqueue to place tasks on for waiting (if any), and the ->done completion flag for indicating whether it's completed or not.

Completions should be named to refer to the event that is being synchronized on. A good example is:

```
wait_for_completion(&early_console_added);

complete(&early_console_added);
```

Good, intuitive naming (as always) helps code readability. Naming a completion 'complete' is not helpful unless the purpose is super obvious...

Initializing completions:

Dynamically allocated completion objects should preferably be embedded in data structures that are assured to be alive for the life-time of the function/driver, to prevent races with asynchronous complete() calls from occurring.

Particular care should be taken when using the _timeout() or _killable()/_interruptible() variants of wait_for_completion(), as it must be assured that memory de-allocation does not happen until all related activities (complete() or reinit_completion()) have taken place, even if these wait functions return prematurely due to a timeout or a signal triggering.

Initializing of dynamically allocated completion objects is done via a call to init_completion():

```
init_completion(&dynamic_object->done);
```

In this call we initialize the waitqueue and set ->done to 0, i.e. "not completed" or "not done".

The re-initialization function, reinit_completion(), simply resets the ->done field to 0 ("not done"), without touching the waitqueue. Callers of this function must make sure that there are no racy wait_for_completion() calls going on in parallel.

Calling init_completion() on the same completion object twice is most likely a bug as it re-initializes the queue to an empty queue and enqueued tasks could get "lost" - use reinit_completion() in that case, but be aware of other races.

For static declaration and initialization, macros are available.

For static (or global) declarations in file scope you can use DECLARE_COMPLETION():

```
static DECLARE_COMPLETION(setup_done);
DECLARE_COMPLETION(setup_done);
```

Note that in this case the completion is boot time (or module load time) initialized to 'not done' and doesn't require an init_completion() call.

When a completion is declared as a local variable within a function, then the initialization should always use DECLARE_COMPLETION_ONSTACK() explicitly, not just to make lockdep happy, but also to make it clear that limited scope had been considered and is intentional:

```
DECLARE_COMPLETION_ONSTACK(setup_done)
```

Note that when using completion objects as local variables you must be acutely aware of the short life time of the function stack: the function must not return to a calling context until all activities (such as waiting threads) have ceased and the completion object is completely unused.

To emphasise this again: in particular when using some of the waiting API variants with more complex outcomes, such as the timeout or signalling (_timeout(), _killable() and _interruptible()) variants, the wait might complete prematurely while the object might still be in use by another thread - and a return from the wait_on_completion*() caller function will deallocate the function stack and cause subtle data corruption if a complete() is done in some other thread. Simple testing might not trigger these kinds of races.

If unsure, use dynamically allocated completion objects, preferably embedded in some other long lived object that has a boringly long life time which exceeds the life time of any helper threads using the completion object, or has a lock or other synchronization mechanism to make sure complete() is not called on a freed object.

A naive DECLARE_COMPLETION() on the stack triggers a lockdep warning.

Waiting for completions:

For a thread to wait for some concurrent activity to finish, it calls wait_for_completion() on the initialized completion structure:

```
void wait_for_completion(struct completion *done)
```

A typical usage scenario is:

```
CPU#1                                   CPU#2

struct completion setup_done;

init_completion(&setup_done);
initialize_work(...,&setup_done,...);

/* run non-dependent code */            /* do setup */

wait_for_completion(&setup_done);       complete(setup_done);
```

This is not implying any particular order between wait_for_completion() and the call to complete() - if the call to complete() happened before the call to wait_for_completion() then the waiting side simply will continue immediately as all dependencies are satisfied; if not, it will block until completion is signaled by complete().

Note that wait_for_completion() is calling spin_lock_irq()/spin_unlock_irq(), so it can only be called safely when you know that interrupts are enabled. Calling it from IRQs-off atomic contexts will result in hard-to-detect spurious enabling of interrupts.

The default behavior is to wait without a timeout and to mark the task as uninterruptible. wait_for_completion() and its variants are only safe in process context (as they can sleep) but not in atomic context, interrupt context, with disabled IRQs, or preemption is disabled - see also try_wait_for_completion() below for handling completion in atomic/interrupt context.

As all variants of wait_for_completion() can (obviously) block for a long time depending on the nature of the activity they are waiting for, so in most cases you probably don't want to call this with held mutexes.

wait_for_completion*() variants available:

The below variants all return status and this status should be checked in most(/all) cases - in cases where the status is deliberately not checked you probably want to make a note explaining this (e.g. see arch/arm/kernel/smp.c:__cpu_up()).

A common problem that occurs is to have unclean assignment of return types, so take care to assign return-values to variables of the proper type.

Checking for the specific meaning of return values also has been found to be quite inaccurate, e.g. constructs like:

```
if (!wait_for_completion_interruptible_timeout(...))
```

... would execute the same code path for successful completion and for the interrupted case - which is probably not what you want:

```
int wait_for_completion_interruptible(struct completion *done)
```

This function marks the task TASK_INTERRUPTIBLE while it is waiting. If a signal was received while waiting it will return -ERESTARTSYS; 0 otherwise:

```
unsigned long wait_for_completion_timeout(struct completion *done, unsigned long timeout)
```

The task is marked as TASK_UNINTERRUPTIBLE and will wait at most 'timeout' jiffies. If a timeout occurs it returns 0, else the remaining time in jiffies (but at least 1).

Timeouts are preferably calculated with msecs_to_jiffies() or usecs_to_jiffies(), to make the code largely HZ-invariant.

If the returned timeout value is deliberately ignored a comment should probably explain why (e.g. see drivers/mfd/wm8350-core.c wm8350_read_auxadc()):

```
long wait_for_completion_interruptible_timeout(struct completion *done, unsigned long timeout)
```

This function passes a timeout in jiffies and marks the task as TASK_INTERRUPTIBLE. If a signal was received it will return -ERESTARTSYS; otherwise it returns 0 if the completion timed out, or the remaining time in jiffies if completion occurred.

Further variants include _killable which uses TASK_KILLABLE as the designated tasks state and will return -ERESTARTSYS if it is interrupted, or 0 if completion was achieved. There is a _timeout variant as well:

```
long wait_for_completion_killable(struct completion *done)
long wait_for_completion_killable_timeout(struct completion *done, unsigned long timeout)
```

The _io variants wait_for_completion_io() behave the same as the non-_io variants, except for accounting waiting time as 'waiting on IO', which has an impact on how the task is accounted in scheduling/IO stats:

```
void wait_for_completion_io(struct completion *done)
unsigned long wait_for_completion_io_timeout(struct completion *done, unsigned long timeout)
```

Signaling completions:

A thread that wants to signal that the conditions for continuation have been achieved calls complete() to signal exactly one of the waiters that it can continue:

```
void complete(struct completion *done)
```

... or calls complete_all() to signal all current and future waiters:

```
void complete_all(struct completion *done)
```

The signaling will work as expected even if completions are signaled before a thread starts waiting. This is achieved by the waiter "consuming" (decrementing) the done field of 'struct completion'. Waiting threads wakeup order is the same in which they were enqueued (FIFO order).

If complete() is called multiple times then this will allow for that number of waiters to continue - each call to complete() will simply increment the done field. Calling complete_all() multiple times is a bug though. Both complete() and complete_all() can be called in IRQ/atomic context safely.

There can only be one thread calling complete() or complete_all() on a particular 'struct completion' at any time - serialized through the wait queue spinlock. Any such concurrent calls to complete() or complete_all() probably are a design bug.

Signaling completion from IRQ context is fine as it will appropriately lock with spin_lock_irqsave()/spin_unlock_irqrestore() and it will never sleep.

try_wait_for_completion()/completion_done():

The try_wait_for_completion() function will not put the thread on the wait queue but rather returns false if it would need to enqueue (block) the thread, else it consumes one posted completion and returns true:

```
bool try_wait_for_completion(struct completion *done)
```

Finally, to check the state of a completion without changing it in any way, call completion_done(), which returns false if there are no posted completions that were not yet consumed by waiters (implying that there are waiters) and true otherwise:

```
bool completion_done(struct completion *done)
```

Both try_wait_for_completion() and completion_done() are safe to be called in IRQ or atomic context.