RCU and lockdep checking

All flavors of RCU have lockdep checking available, so that lockdep is aware of when each task enters and leaves any flavor of RCU read-side critical section. Each flavor of RCU is tracked separately (but note that this is not the case in 2.6.32 and earlier). This allows lockdep's tracking to include RCU state, which can sometimes help when debugging deadlocks and the like.

In addition, RCU provides the following primitives that check lockdep's state:

```
rcu_read_lock_held() for normal RCU.
rcu_read_lock_bh_held() for RCU-bh.
rcu_read_lock_sched_held() for RCU-sched.
srcu_read_lock_held() for SRCU.
```

These functions are conservative, and will therefore return 1 if they aren't certain (for example, if CONFIG_DEBUG_LOCK_ALLOC is not set). This prevents things like WARN_ON(!rcu_read_lock_held()) from giving false positives when lockdep is disabled.

In addition, a separate kernel config parameter CONFIG_PROVE_RCU enables checking of rcu_dereference() primitives:

```
rcu_dereference(p):
        Check for RCU read-side critical section.
rcu_dereference_bh(p):
        Check for RCU-bh read-side critical section.
rcu_dereference_sched(p):
        Check for RCU-sched read-side critical section.
srcu_dereference(p, sp):
        Check for SRCU read-side critical section.
rcu_dereference_check(p, c):
        Use explicit check expression "c". This is useful in
        code that is invoked by both readers and updaters.
rcu_dereference_raw(p)
        Don't check. (Use sparingly, if at all.)
rcu_dereference_protected(p, c):
        Use explicit check expression "c", and omit all barriers
        and compiler constraints. This is useful when the data
        structure cannot change, for example, in code that is
        invoked only by updaters.
rcu_access_pointer(p):
        Return the value of the pointer and omit all barriers,
        but retain the compiler constraints that prevent duplicating
        or coalescsing. This is useful when when testing the
        value of the pointer itself, for example, against NULL.
```

The rcu_dereference_check() check expression can be any boolean expression, but would normally include one of the rcu_read_lock_held() family of functions and a lockdep expression. However, any boolean expression can be used. For a moderately ornate example, consider the following:

```
file = rcu_dereference_check(fdt->fd[fd],
```

```
                                rcu_read_lock_held() ||
                                lockdep_is_held(&files->file_lock) ||
                                atomic_read(&files->count) == 1);
```

This expression picks up the pointer "fdt->fd[fd]" in an RCU-safe manner, and, if CONFIG_PROVE_RCU is configured, verifies that this expression is used in:

1. An RCU read-side critical section, or
2. with files->file_lock held, or
3. on an unshared files_struct.

In case (1), the pointer is picked up in an RCU-safe manner for vanilla RCU read-side critical sections, in case (2) the ->file_lock prevents any change from taking place, and finally, in case (3) the current task is the only task accessing the file_struct, again preventing any change from taking place. If the above statement was invoked only from updater code, it could instead be written as follows:

```
        file = rcu_dereference_protected(fdt->fd[fd],
                                         lockdep_is_held(&files->file_lock) ||
                                         atomic_read(&files->count) == 1);
```

This would verify cases #2 and #3 above, and furthermore lockdep would complain if this was used in an RCU read-side critical section unless one of these two cases held. Because rcu_dereference_protected() omits all barriers and compiler constraints, it generates better code than do the other flavors of rcu_dereference(). On the other hand, it is illegal to use rcu_dereference_protected() if either the RCU-protected pointer or the RCU-protected data that it points to can change concurrently.

There are currently only "universal" versions of the rcu_assign_pointer() and RCU list-/tree-traversal primitives, which do not (yet) check for being in an RCU read-side critical section. In the future, separate versions of these primitives might be created.