RCU Tasks & PID-Namespace Unshare

This document looks at an entertaining deadlock scenario reported by Pengfei Xu.

Deadlock Scenario

RCU Tasks and PID-namespace unshare can interact in `do_exit()` in a complicated circular dependency, as analyzed by Frederic Weisbecker:

1. TASK A calls unshare(CLONE_NEWPID), this creates a new PID namespace that every subsequent child of TASK A will belong to. But TASK A doesn't itself belong to that new PID namespace.
2. TASK A forks() and creates TASK B (it is a new thread group so it is a thread group leader). TASK A stays attached to its PID namespace (let's say PID_NS1) and TASK B is the first task belonging to the new PID namespace created by unshare() (let's call it PID_NS2).
3. Since TASK B is the first task attached to PID_NS2, it becomes the PID_NS2 child reaper.
4. TASK A forks() again and creates TASK C which get attached to PID_NS2. Note how TASK C has TASK A as a parent (belonging to PID_NS1) but has TASK B (belonging to PID_NS2) as a pid_namespace child_reaper.
5. TASK B exits and since it is the child reaper for PID_NS2, it has to kill all other tasks attached to PID_NS2, and wait for all of them to die before reaping itself (zap_pid_ns_process()). Note it seems to make a misleading assumption here, trusting that all tasks in PID_NS2 either get reaped by a parent belonging to the same namespace or by TASK B. And it is confident that since it deactivated SIGCHLD handler, all the remaining tasks ultimately autoreap. And it waits for that to happen. However TASK C escapes that rule because it will get reaped by its parent TASK A belonging to PID_NS1.
6. TASK A calls synchronize_rcu_tasks() which leads to synchronize_srcu(&tasks_rcu_exit_srcu).
7. TASK B is waiting for TASK C to get reaped (wrongly assuming it autoreaps) But TASK B is under a tasks_rcu_exit_srcu SRCU critical section (exit_notify() is between exit_tasks_rcu_start() and exit_tasks_rcu_finish()), blocking TASK A
8. TASK C exits and since TASK A is its parent, it waits for it to reap TASK C, but it can't because TASK A waits for TASK B that waits for TASK C.

So there is a circular dependency, also known as deadlock:

- TASK A waits for TASK B to get out of tasks_rcu_exit_srcu SRCU critical section
- TASK B waits for TASK C to get reaped
- TASK C waits for TASK A to reap it.

I have no idea how to solve the situation without violating the pid_namespace rules and unshare() semantics (although I wish unshare(CLONE_NEWPID) had a less error prone behaviour with allowing creating more than one task belonging to the same namespace).

So probably having an SRCU read side critical section within exit_notify() is not a good idea, is there a solution to work around that for rcu tasks?

Reproducer

https://github.com/xupengfe/syzkaller_logs/tree/main/221115_105658_synchronize_rcu

Questions

What is the RCU Tasks Task-Exit Rationale?

Neeraj Upadhyay notes that this topic is touched upon in Inspection of RCU Tasks Trace in question 3 (“Use of synchronize_srcu() in rcu_tasks_postscan() and in rcu_tasks_postgp()”). The commit log for `3f95aa81d265 ("rcu: Make TASKS_RCU handle tasks that are almost done exiting")` is quite forthright about this issue:

> Once a task has passed `exit_notify()` in the `do_exit()` code path, it is no longer on the task lists, and is therefore no longer visible to `rcu_tasks_kthread()`.

The high-level issue is that tracers can be attached to functions very late in `do_exit()`. It is therefore necessary for `synchronize_rcu_tasks()` to wait for the dying task to execute its last instruction. Even though it is no longer on the tasks lists. Therefore, do_exit() calls

One complication is that the task removes itself from tasks list, beyond which point `synchronize_rcu_tasks()` cannot see that task via the task-list scan. Therefore, there must be some mechanism to wait for tasks that have already been removed from the tasks list, but which have not yet executed their very last instruction. This mechanism is SRCU, as put in place by the above commit, but as of 2022 using `exit_tasks_rcu_start() and exit_tasks_rcu_finish()` wrapper functions for `srcu_read_lock()` and `srcu_read_unlock()`.

Unfortunately, this results in the deadlock called out above.

Why Disable Preemption in `exit_tasks_rcu_start()` and `exit_tasks_rcu_finish()`?

Boqun Feng notes that `srcu_read_lock()` and `srcu_read_unlock()` used to disable preemption, but that a later commit removed the need to do so. When SRCU switched to a simpler read-side algorithm that featured this_cpu_inc(), the preemption disabling was removed from , but `exit_tasks_rcu_start() and exit_tasks_rcu_finish()` never were updated.

Therefore, the `preempt_disable()` and `preempt_enable()` calls can be safely removed from the `exit_tasks_rcu_start() and exit_tasks_rcu_finish()` functions.

Why use __srcu_read_lock() instead of srcu_read_lock()?

It appears that back when the SRCU read-side critical section was introduced, there was a lockdep false positive: https://lore.kernel.org/lkml/20140811224840.GA25594@linux.vnet.ibm.com/

> Moved from srcu_read_lock() and srcu_read_unlock to __srcu_read_lock()
> and __srcu_read_unlock() to avoid CONFIG_PROVE_RCU false positives
> in do_exit().

What Are Possible Fixes?

Frederic Weisbecker suggests exiting and re-entering the SRCU read-side critical section. However, exiting that critical section by calling `exit_tasks_rcu_finish()` would be bad because that would invoke `exit_tasks_rcu_finish_trace()` which could prematurely end an RCU Tasks Trace read-side critical section. Perhaps there should be an `exit_tasks_rcu_pause()` that is invoked by `exit_tasks_rcu_finish()`?

Note that the SRCU reader index is maintained in `current->rcu_tasks_idx`, so that issue is handled.

Potential diagnostics:

- Perhaps kernels built with lockdep should set a timer and complain if it takes too long to get through `do_exit()`. After all, there might be other similar deadlocks. However, what should that timeout value be?
- It would be good for RCU Tasks and RCU Tasks Trace RCU CPU stall warnings to cover the `synchronize_rcu()` and `synchronize_srcu()` invocations. (Neeraj is looking into this.)
- Expand the comment preceding the `synchronize_srcu()` to explicitly state that part of the problem to be solved is that tasks are removed from the task list before they

disable preemption, and that synchronize_rcu_tasks() must wait for an exiting task to execute its last instruction (which is why there is the check for `t->on_rq` in `check_holdout_task()`).

Note well that none of this addresses the namespace-leak problem that can be caused by only slightly different use cases. Risks notwithstanding, it might prove worthwhile to revisit namespace semantics.