Userspace RCU - Bug #991

Memory leak after call_rcu

01/22/2016 03:56 AM - Daniel Salzman

Status:	Resolved	**Start date:**	01/22/2016
Priority:	Normal	**Due date:**	
Assignee:		**% Done:**	0%
Category:		**Estimated time:**	0.00 hour
Target version:			

Description

The problem also applies to the latest 0.9.1 source. Attached a valgrind summary
for doc/examples/urcu-flavors/mb.c

```
16229 288 bytes in 1 blocks are possibly lost in loss record 2 of 2
16229    at 0x4C2CC70: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
16229    by 0x4012E54: allocate_dtv (dl-tls.c:296)
16229    by 0x4012E54: _dl_allocate_tls (dl-tls.c:460)
16229    by 0x5046DA0: allocate_stack (allocatestack.c:589)
16229    by 0x5046DA0: pthread_create@@GLIBC_2.2.5 (pthread_create.c:500)
16229    by 0x4E3947F: call_rcu_data_init (urcu-call-rcu-impl.h:435)
16229    by 0x4E3A742: get_default_call_rcu_data_mb (urcu-call-rcu-impl.h:562)
16229    by 0x4E3A7D7: call_rcu_mb (urcu-call-rcu-impl.h:712)
16229    by 0x400AFA: main (mb.c:117)
16229
16229 LEAK SUMMARY:
16229    definitely lost: 0 bytes in 0 blocks
16229    indirectly lost: 0 bytes in 0 blocks
16229      possibly lost: 288 bytes in 1 blocks
16229    still reachable: 128 bytes in 1 blocks
16229         suppressed: 0 bytes in 0 blocks
```

History

#1 - 07/15/2016 02:27 PM - Mathieu Desnoyers

Let me give a bit of background on the call_rcu mechanism to explain this leak.

Basically, when call_rcu is used, it enqueues an application-provided callbacks to be executed later (after a grace period) by a call_rcu worker thread. Then, when the callback is executed, it can e.g. free some memory (typical use-case), but it can also choose to re-enqueue more callbacks.

This is because of this chaining that we cannot ever guarantee to reach a "quiescent state" where we can free the default call_rcu work queue and worker thread on process teardown.

AFAIK, the Linux kernel deals with this in a similar fashion: if there is still work enqueued when the machine resets, the actual reset takes care of wiping out all resources.

The equivalent in user-space is to let teardown of the process (performed by the kernel) handle this, but it leaves a "leak" of the default call_rcu queue and worker thread.

Perhaps we should consider adding this to a Valgrind whitelist ?

#2 - 11/30/2016 09:08 PM - Mathieu Desnoyers

- Status changed from New to Feedback

#3 - 12/01/2016 08:04 AM - Daniel Salzman

Hi Mathieu,
What about adding something like "rcu_cleanup" which, if called, will free the call_rcu queue explicitly?

#4 - 01/30/2017 12:13 PM - Mathieu Desnoyers

With this explicit rcu_cleanup scheme, how do you propose we handle cases where there are still work items in the queue when rcu_cleanup is invoked ?

#5 - 01/30/2017 01:18 PM - Daniel Salzman

If rcu_cleanup could block until the queue is not empty, then we would be happy :-)

#6 - 01/30/2017 01:40 PM - Mathieu Desnoyers

Then what should be the behavior when the following scenario occurs ?

call_rcu enqueues a callback, which itself invokes call_rcu to enqueue the same callback, repeatedly. This will cause the call_rcu queue to never be empty.

With the proposed rcu_cleanup behavior, it would hang the application forever on rcu_cleanup(). This seems to be an unwanted side-effect.

#7 - 01/30/2017 02:10 PM - Daniel Salzman

I understand this argument.

It is important to say that invoking of "rcu_cleanup" is optional. Just a hint for the library that the program ensures such "safe" conditions.

#8 - 01/30/2017 02:20 PM - Daniel Salzman

The reason why we don't like this leak is our automated testing, which is based on Valgrind and would need some workaround if we used call_rcu.

#9 - 01/30/2017 03:55 PM - Mathieu Desnoyers

Perhaps we could introduce a "call_rcu_try_cleanup()", which could
succeed if there are no callbacks currently queued in any of the
call_rcu queues (including the default queue), and fail otherwise.

It would be up to the application to control what it does with call_rcu
callbacks and use rcu_barrier() appropriately in order to ensure it
does succeed, or deal with failure in the way it find appropriate.

Thoughts ?

Thanks,

Mathieu

#10 - 01/30/2017 04:15 PM - Daniel Salzman

Yes, that sound good.

Thank you!

#11 - 01/30/2017 08:48 PM - Mathieu Desnoyers

Well this gives us a possible API. Now let's consider what happens if we add this API.

We have call_rcu APIs linked to each urcu flavor (urcu-bp, urcu-mb, urcu, urcu-signal, urcu-qsbr).

Let's take urcu-bp for instance. urcu-bp is expected to track RCU reader threads (does not require explicit thread registration), and libraries can use urcu-bp without requiring the application to know about it (use-case: tracing library).

So this is a case where a library may still use urcu-bp even when the application main() exits: library destructors.

Therefore, having the application call a call_rcu_try_cleanup() does not appear to be appropriate for the library use-case: a library may still be actively using the call_rcu data structures concurrently with this call_rcu_try_cleanup(), which could trigger a failure.

One alternative possibility would be to use constructors, destructors, and reference counting to deal with all this, but this is becoming very elaborate to simply silence a Valgrind warning about a variable that is reclaimed by the OS.

Valgrind has the concept of "whitelist", which I think we should use for this particular call_rcu data structure, and let the OS perform the reclaim.

#12 - 01/30/2017 10:02 PM - Daniel Salzman

Ok, you have convinced me. Thank you for your time and effort! You can close the issue.

#13 - 01/30/2017 10:10 PM - Jonathan Rajotte Julien

- Status changed from Feedback to Resolved