RCU Torture Test Operation

CONFIG_RCU_TORTURE_TEST

The CONFIG_RCU_TORTURE_TEST config option is available for all RCU implementations. It creates an rcutorture kernel module that can be loaded to run a torture test. The test periodically outputs status messages via printk(), which can be examined via the dmesg command (perhaps grepping for "torture"). The test is started when the module is loaded, and stops when the module is unloaded.

Module parameters are prefixed by "rcutorture." in Documentation/admin-guide/kernel-parameters.txt.

Output

The statistics output is as follows:

```
rcu-torture:--- Start of test: nreaders=16 nfakewriters=4 stat_interval=30 verbose=0 test_no_idle_hz=1 shuffle_interval=3 stutter=5 irqre
rcu-torture: rtc:           (null) ver: 155441 tfle: 0 rta: 155441 rtaf: 8884 rtf: 155440 rtmbe: 0 rtbe: 0 rtbke: 0 rtbre: 0 rtbf: 0 rtb
rcu-torture: Reader Pipe:  727860534 34213 0 0 0 0 0 0 0 0 0
rcu-torture: Reader Batch:  727877838 17003 0 0 0 0 0 0 0 0 0
rcu-torture: Free-Block Circulation:  155440 155440 155440 155440 155440 155440 155440 155440 155440 155440 0
rcu-torture:--- End of test: SUCCESS: nreaders=16 nfakewriters=4 stat_interval=30 verbose=0 test_no_idle_hz=1 shuffle_interval=3 stutter=
```

The command "dmesg | grep torture:" will extract this information on most systems. On more esoteric configurations, it may be necessary to use other commands to access the output of the printk()s used by the RCU torture test. The printk()s use KERN_ALERT, so they should be evident. ;-)

The first and last lines show the rcutorture module parameters, and the last line shows either "SUCCESS" or "FAILURE", based on rcutorture's automatic determination as to whether RCU operated correctly.

The entries are as follows:

- "rtc": The hexadecimal address of the structure currently visible to readers.
- "ver": The number of times since boot that the RCU writer task has changed the structure visible to readers.
- "tfle": If non-zero, indicates that the "torture freelist" containing structures to be placed into the "rtc" area is empty. This condition is important, since it can fool you into thinking that RCU is working when it is not. :-/
- "rta": Number of structures allocated from the torture freelist.
- "rtaf": Number of allocations from the torture freelist that have failed due to the list being empty. It is not unusual for this to be non-zero, but it is bad for it to be a large fraction of the value indicated by "rta".
- "rtf": Number of frees into the torture freelist.
- "rtmbe": A non-zero value indicates that rcutorture believes that rcu_assign_pointer() and rcu_dereference() are not working correctly. This value should be zero.
- "rtbe": A non-zero value indicates that one of the rcu_barrier() family of functions is not working correctly.
- "rtbke": rcutorture was unable to create the real-time kthreads used to force RCU priority inversion. This value should be zero.
- "rtbre": Although rcutorture successfully created the kthreads used to force RCU priority inversion, it was unable to set them to the real-time priority level of 1. This value should be zero.
- "rtbf": The number of times that RCU priority boosting failed to resolve RCU priority inversion.
- "rtb": The number of times that rcutorture attempted to force an RCU priority inversion condition. If you are testing RCU priority boosting via the "test_boost" module parameter, this value should be non-zero.
- "nt": The number of times rcutorture ran RCU read-side code from within a timer handler. This value should be non-zero only if you specified the "irqreader" module parameter.
- "Reader Pipe": Histogram of "ages" of structures seen by readers. If any entries past the first two are non-zero, RCU is broken. And rcutorture prints the error flag string "!!!" to make sure you notice. The age of a newly allocated structure is zero, it becomes one when removed from reader visibility, and is incremented once per grace period subsequently -- and is freed after passing through (RCU_TORTURE_PIPE_LEN-2) grace periods.

  The output displayed above was taken from a correctly working RCU. If you want to see what it looks like when broken, break it yourself. ;-)
- "Reader Batch": Another histogram of "ages" of structures seen by readers, but in terms of counter flips (or batches) rather than in terms of grace periods. The legal number of non-zero entries is again two. The reason for this separate view is that it is sometimes easier to get the third entry to show up in the "Reader Batch" list than in the "Reader Pipe" list.
- "Free-Block Circulation": Shows the number of torture structures that have reached a given point in the pipeline. The first element should closely correspond to the number of structures allocated, the second to the number that have been removed from reader view, and all but the last remaining to the corresponding number of passes through a grace period. The last entry should be zero, as it is only incremented if a torture structure's counter somehow gets incremented farther than it should.

Different implementations of RCU can provide implementation-specific additional information. For example, Tree SRCU provides the following additional line:

```
srcud-torture: Tree SRCU per-CPU(idx=0): 0(35,-21) 1(-4,24) 2(1,1) 3(-26,20) 4(28,-47) 5(-9,4) 6(-10,14) 7(-14,11) T(1,6)
```

This line shows the per-CPU counter state, in this case for Tree SRCU using a dynamically allocated srcu_struct (hence "srcud-" rather than "srcu-"). The numbers in parentheses are the values of the "old" and "current" counters for the corresponding CPU. The "idx" value maps the "old" and "current" values to the underlying array, and is useful for debugging. The final "T" entry contains the totals of the counters.

Usage on Specific Kernel Builds

It is sometimes desirable to torture RCU on a specific kernel build, for example, when preparing to put that kernel build into production. In that case, the kernel should be built with CONFIG_RCU_TORTURE_TEST=m so that the test can be started using modprobe and terminated using rmmod.

For example, the following script may be used to torture RCU:

```
#!/bin/sh

modprobe rcutorture
sleep 3600
rmmod rcutorture
dmesg | grep torture:
```

The output can be manually inspected for the error flag of "!!!". One could of course create a more elaborate script that automatically checked for such errors. The "rmmod" command forces a "SUCCESS", "FAILURE", or "RCU_HOTPLUG" indication to be printk()ed. The first two are self-explanatory, while the last indicates that while there were no RCU failures, CPU-hotplug problems were detected.

Usage on Mainline Kernels

When using rcutorture to test changes to RCU itself, it is often necessary to build a number of kernels in order to test that change across a broad range of combinations of the relevant Kconfig options and of the relevant kernel boot parameters. In this situation, use of modprobe and rmmod can be quite time-consuming and error-prone.

Therefore, the tools/testing/selftests/rcutorture/bin/kvm.sh script is available for mainline testing for x86, arm64, and powerpc. By default, it will run the series of tests specified by tools/testing/selftests/rcutorture/configs/rcu/CFLIST, with each test running for 30 minutes within a guest OS using a minimal userspace supplied by an automatically generated initrd. After the tests are complete, the resulting build products and console output are analyzed for errors and the results of the runs are summarized.

On larger systems, rcutorture testing can be accelerated by passing the --cpus argument to kvm.sh. For example, on a 64-CPU system, "--cpus 43" would use up to 43 CPUs to run tests concurrently, which as of v5.4 would complete all the scenarios in two batches, reducing the time to complete from about eight hours to about one hour (not counting the time to build the sixteen kernels). The "--dryrun sched" argument will not run tests, but rather tell you how the tests would be scheduled into batches. This can be useful when working out how many CPUs to specify in the --cpus argument.

Not all changes require that all scenarios be run. For example, a change to Tree SRCU might run only the SRCU-N and SRCU-P scenarios using the --configs argument to kvm.sh as follows: "--configs 'SRCU-N SRCU-P'". Large systems can run multiple copies of of the full set of scenarios, for example, a system with 448 hardware threads can run five instances of the full set concurrently. To make this happen:

```
kvm.sh --cpus 448 --configs '5*CFLIST'
```

Alternatively, such a system can run 56 concurrent instances of a single eight-CPU scenario:

```
kvm.sh --cpus 448 --configs '56*TREE04'
```

Or 28 concurrent instances of each of two eight-CPU scenarios:

```
kvm.sh --cpus 448 --configs '28*TREE03 28*TREE04'
```

Of course, each concurrent instance will use memory, which can be limited using the --memory argument, which defaults to 512M. Small values for memory may require disabling the callback-flooding tests using the --bootargs parameter discussed below.

Sometimes additional debugging is useful, and in such cases the --kconfig parameter to kvm.sh may be used, for example, `--kconfig 'CONFIG_KASAN=y'`.

Kernel boot arguments can also be supplied, for example, to control rcutorture's module parameters. For example, to test a change to RCU's CPU stall-warning code, use "--bootargs 'rcutorture.stall_cpu=30'". This will of course result in the scripting reporting a failure, namely the resuling RCU CPU stall warning. As noted above, reducing memory may require disabling rcutorture's callback-flooding tests:

```
kvm.sh --cpus 448 --configs '56*TREE04' --memory 128M \
       --bootargs 'rcutorture.fwd_progress=0'
```

Sometimes all that is needed is a full set of kernel builds. This is what the --buildonly argument does.

Finally, the --trust-make argument allows each kernel build to reuse what it can from the previous kernel build.

There are additional more arcane arguments that are documented in the source code of the kvm.sh script.

If a run contains failures, the number of buildtime and runtime failures is listed at the end of the kvm.sh output, which you really should redirect to a file. The build products and console output of each run is kept in tools/testing/selftests/rcutorture/res in timestamped directories. A given directory can be supplied to kvm-find-errors.sh in order to have it cycle you through summaries of errors and full error logs. For example:

```
tools/testing/selftests/rcutorture/bin/kvm-find-errors.sh \
        tools/testing/selftests/rcutorture/res/2020.01.20-15.54.23
```

However, it is often more convenient to access the files directly. Files pertaining to all scenarios in a run reside in the top-level directory (2020.01.20-15.54.23 in the example above), while per-scenario files reside in a subdirectory named after the scenario (for example, "TREE04"). If a given scenario ran more than once (as in "--configs '56*TREE04'" above), the directories corresponding to the second and subsequent runs of that scenario include a sequence number, for example, "TREE04.2", "TREE04.3", and so on.

The most frequently used file in the top-level directory is testid.txt. If the test ran in a git repository, then this file contains the commit that was tested and any uncommitted changes in diff format.

The most frequently used files in each per-scenario-run directory are:

.config:
: This file contains the Kconfig options.

Make.out:
: This contains build output for a specific scenario.

console.log:
: This contains the console output for a specific scenario. This file may be examined once the kernel has booted, but it might not exist if the build failed.

vmlinux:
: This contains the kernel, which can be useful with tools like objdump and gdb.

A number of additional files are available, but are less frequently used. Many are intended for debugging of rcutorture itself or of its scripting.

As of v5.4, a successful run with the default set of scenarios produces the following summary at the end of the run on a 12-CPU system:

```
SRCU-N ------- 804233 GPs (148.932/s) [srcu: g10008272 f0x0 ]
SRCU-P ------- 202320 GPs (37.4667/s) [srcud: g1809476 f0x0 ]
SRCU-t ------- 1122086 GPs (207.794/s) [srcu: g0 f0x0 ]
SRCU-u ------- 1111285 GPs (205.794/s) [srcud: g1 f0x0 ]
TASKS01 ------- 19666 GPs (3.64185/s) [tasks: g0 f0x0 ]
TASKS02 ------- 20541 GPs (3.80389/s) [tasks: g0 f0x0 ]
TASKS03 ------- 19416 GPs (3.59556/s) [tasks: g0 f0x0 ]
TINY01 ------- 836134 GPs (154.84/s) [rcu: g0 f0x0 ] n_max_cbs: 34198
TINY02 ------- 850371 GPs (157.476/s) [rcu: g0 f0x0 ] n_max_cbs: 2631
TREE01 ------- 162625 GPs (30.1157/s) [rcu: g1124169 f0x0 ]
TREE02 ------- 333003 GPs (61.6672/s) [rcu: g2647753 f0x0 ] n_max_cbs: 35844
TREE03 ------- 306623 GPs (56.782/s) [rcu: g2975325 f0x0 ] n_max_cbs: 1496497
CPU count limited from 16 to 12
TREE04 ------- 246149 GPs (45.5831/s) [rcu: g1695737 f0x0 ] n_max_cbs: 434961
TREE05 ------- 314603 GPs (58.2598/s) [rcu: g2257741 f0x2 ] n_max_cbs: 193997
TREE07 ------- 167347 GPs (30.9902/s) [rcu: g1079021 f0x0 ] n_max_cbs: 478732
CPU count limited from 16 to 12
TREE09 ------- 752238 GPs (139.303/s) [rcu: g13075057 f0x0 ] n_max_cbs: 99011
```