The Kernel Concurrency Sanitizer (KCSAN)

The Kernel Concurrency Sanitizer (KCSAN) is a dynamic race detector, which relies on compile-time instrumentation, and uses a watchpoint-based sampling approach to detect races. KCSAN's primary purpose is to detect data races.

Usage

KCSAN is supported by both GCC and Clang. With GCC we require version 11 or later, and with Clang also require version 11 or later.

To enable KCSAN configure the kernel with:

```
CONFIG_KCSAN = y
```

KCSAN provides several other configuration options to customize behaviour (see the respective help text in `lib/Kconfig.kcsan` for more info).

Error reports

A typical data race report looks like this:

```
==================================================================
BUG: KCSAN: data-race in test_kernel_read / test_kernel_write

write to 0xffffffffc009a628 of 8 bytes by task 487 on cpu 0:
 test_kernel_write+0x1d/0x30
 access_thread+0x89/0xd0
 kthread+0x23e/0x260
 ret_from_fork+0x22/0x30

read to 0xffffffffc009a628 of 8 bytes by task 488 on cpu 6:
 test_kernel_read+0x10/0x20
 access_thread+0x89/0xd0
 kthread+0x23e/0x260
 ret_from_fork+0x22/0x30

value changed: 0x00000000000009a6 -> 0x00000000000009b2

Reported by Kernel Concurrency Sanitizer on:
CPU: 6 PID: 488 Comm: access_thread Not tainted 5.12.0-rc2+ #1
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.14.0-2 04/01/2014
==================================================================
```

The header of the report provides a short summary of the functions involved in the race. It is followed by the access types and stack traces of the 2 threads involved in the data race. If KCSAN also observed a value change, the observed old value and new value are shown on the "value changed" line respectively.

The other less common type of data race report looks like this:

```
==================================================================
BUG: KCSAN: data-race in test_kernel_rmw_array+0x71/0xd0

race at unknown origin, with read to 0xffffffffc009bdb0 of 8 bytes by task 515 on cpu 2:
 test_kernel_rmw_array+0x71/0xd0
 access_thread+0x89/0xd0
 kthread+0x23e/0x260
 ret_from_fork+0x22/0x30

value changed: 0x0000000000002328 -> 0x0000000000002329

Reported by Kernel Concurrency Sanitizer on:
CPU: 2 PID: 515 Comm: access_thread Not tainted 5.12.0-rc2+ #1
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.14.0-2 04/01/2014
==================================================================
```

This report is generated where it was not possible to determine the other racing thread, but a race was inferred due to the data value of the watched memory location having changed. These reports always show a "value changed" line. A common reason for reports of this type are missing instrumentation in the racing thread, but could also occur due to e.g. DMA accesses. Such reports are shown only if `CONFIG_KCSAN_REPORT_RACE_UNKNOWN_ORIGIN=y`, which is enabled by default.

Selective analysis

It may be desirable to disable data race detection for specific accesses, functions, compilation units, or entire subsystems. For static blacklisting, the below options are available:

- KCSAN understands the `data_race(expr)` annotation, which tells KCSAN that any data races due to accesses in `expr` should be ignored and resulting behaviour when encountering a data race is deemed safe. Please see "Marking Shared-Memory Accesses" in the LKMM for more information.
- Disabling data race detection for entire functions can be accomplished by using the function attribute `__no_kcsan`:

  ```
  __no_kcsan
  void foo(void) {
      ...
  ```

  To dynamically limit for which functions to generate reports, see the DebugFS interface blacklist/whitelist feature.
- To disable data race detection for a particular compilation unit, add to the `Makefile`:

  ```
  KCSAN_SANITIZE_file.o := n
  ```
- To disable data race detection for all compilation units listed in a `Makefile`, add to the respective `Makefile`:

  ```
  KCSAN_SANITIZE := n
  ```

Furthermore, it is possible to tell KCSAN to show or hide entire classes of data races, depending on preferences. These can be changed via the following Kconfig options:

- `CONFIG_KCSAN_REPORT_VALUE_CHANGE_ONLY`: If enabled and a conflicting write is observed via a watchpoint, but the data value of the memory location was observed to remain unchanged, do not report the data race.
- `CONFIG_KCSAN_ASSUME_PLAIN_WRITES_ATOMIC`: Assume that plain aligned writes up to word size are atomic by default. Assumes that such writes are not subject to unsafe compiler optimizations resulting in data races. The option causes KCSAN to not report data races due to conflicts where the only plain accesses are aligned writes up to word size.
- `CONFIG_KCSAN_PERMISSIVE`: Enable additional permissive rules to ignore certain classes of common data races. Unlike the above, the rules are more complex involving value-change patterns, access type, and address. This option depends on `CONFIG_KCSAN_REPORT_VALUE_CHANGE_ONLY=y`. For details please see the `kernel/kcsan/permissive.h`. Testers and maintainers that only focus on reports from specific subsystems and not the whole kernel are recommended to disable this option.

To use the strictest possible rules, select `CONFIG_KCSAN_STRICT=y`, which configures KCSAN to follow the Linux-kernel memory consistency model (LKMM) as closely as possible.

DebugFS interface

The file `/sys/kernel/debug/kcsan` provides the following interface:

- Reading `/sys/kernel/debug/kcsan` returns various runtime statistics.
- Writing `on` or `off` to `/sys/kernel/debug/kcsan` allows turning KCSAN on or off, respectively.
- Writing `!some_func_name` to `/sys/kernel/debug/kcsan` adds `some_func_name` to the report filter list, which (by default) blacklists reporting data races where either one of the top stackframes are a function in the list.
- Writing either `blacklist` or `whitelist` to `/sys/kernel/debug/kcsan` changes the report filtering behaviour. For example, the blacklist feature can be used to silence frequently occurring data races; the whitelist feature can help with reproduction and testing of fixes.

Tuning performance

Core parameters that affect KCSAN's overall performance and bug detection ability are exposed as kernel command-line arguments whose defaults can also be changed via the corresponding Kconfig options.

- `kcsan.skip_watch` (`CONFIG_KCSAN_SKIP_WATCH`): Number of per-CPU memory operations to skip, before another watchpoint is set up. Setting up watchpoints more frequently will result in the likelihood of races to be observed to increase. This parameter has the most significant impact on overall system performance and race detection ability.
- `kcsan.udelay_task` (`CONFIG_KCSAN_UDELAY_TASK`): For tasks, the microsecond delay to stall execution after a watchpoint has been set up. Larger values result in the window in which we may observe a race to increase.
- `kcsan.udelay_interrupt` (`CONFIG_KCSAN_UDELAY_INTERRUPT`): For interrupts, the microsecond delay to stall execution after a watchpoint has been set up. Interrupts have tighter latency requirements, and their delay should generally be smaller than the one chosen for tasks.

They may be tweaked at runtime via `/sys/module/kcsan/parameters/`.

Data Races

In an execution, two memory accesses form a *data race* if they *conflict*, they happen concurrently in different threads, and at least one of them is a *plain access*; they *conflict* if both access the same memory location, and at least one is a write. For a more thorough discussion and definition, see "Plain Accesses and Data Races" in the LKMM.

Relationship with the Linux-Kernel Memory Consistency Model (LKMM)

The LKMM defines the propagation and ordering rules of various memory operations, which gives developers the ability to reason

about concurrent code. Ultimately this allows to determine the possible executions of concurrent code, and if that code is free from data races.

KCSAN is aware of *marked atomic operations* (`READ_ONCE`, `WRITE_ONCE`, `atomic_*`, etc.), and a subset of ordering guarantees implied by memory barriers. With `CONFIG_KCSAN_WEAK_MEMORY=y`, KCSAN models load or store buffering, and can detect missing `smp_mb()`, `smp_wmb()`, `smp_rmb()`, `smp_store_release()`, and all `atomic_*` operations with equivalent implied barriers.

Note, KCSAN will not report all data races due to missing memory ordering, specifically where a memory barrier would be required to prohibit subsequent memory operation from reordering before the barrier. Developers should therefore carefully consider the required memory ordering requirements that remain unchecked.

Race Detection Beyond Data Races

For code with complex concurrency design, race-condition bugs may not always manifest as data races. Race conditions occur if concurrently executing operations result in unexpected system behaviour. On the other hand, data races are defined at the C-language level. The following macros can be used to check properties of concurrent code where bugs would not manifest as data races.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\(linux-master)(Documentation)(dev-tools)kcsan.rst`, line 228)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/kcsan-checks.h
    :functions: ASSERT_EXCLUSIVE_WRITER ASSERT_EXCLUSIVE_WRITER_SCOPED
                ASSERT_EXCLUSIVE_ACCESS ASSERT_EXCLUSIVE_ACCESS_SCOPED
                ASSERT_EXCLUSIVE_BITS
```

Implementation Details

KCSAN relies on observing that two accesses happen concurrently. Crucially, we want to (a) increase the chances of observing races (especially for races that manifest rarely), and (b) be able to actually observe them. We can accomplish (a) by injecting various delays, and (b) by using address watchpoints (or breakpoints).

If we deliberately stall a memory access, while we have a watchpoint for its address set up, and then observe the watchpoint to fire, two accesses to the same address just raced. Using hardware watchpoints, this is the approach taken in DataCollider. Unlike DataCollider, KCSAN does not use hardware watchpoints, but instead relies on compiler instrumentation and "soft watchpoints".

In KCSAN, watchpoints are implemented using an efficient encoding that stores access type, size, and address in a long; the benefits of using "soft watchpoints" are portability and greater flexibility. KCSAN then relies on the compiler instrumenting plain accesses. For each instrumented plain access:

1. Check if a matching watchpoint exists; if yes, and at least one access is a write, then we encountered a racing access.
2. Periodically, if no matching watchpoint exists, set up a watchpoint and stall for a small randomized delay.
3. Also check the data value before the delay, and re-check the data value after delay; if the values mismatch, we infer a race of unknown origin.

To detect data races between plain and marked accesses, KCSAN also annotates marked accesses, but only to check if a watchpoint exists; i.e. KCSAN never sets up a watchpoint on marked accesses. By never setting up watchpoints for marked operations, if all accesses to a variable that is accessed concurrently are properly marked, KCSAN will never trigger a watchpoint and therefore never report the accesses.

Modeling Weak Memory

KCSAN's approach to detecting data races due to missing memory barriers is based on modeling access reordering (with `CONFIG_KCSAN_WEAK_MEMORY=y`). Each plain memory access for which a watchpoint is set up, is also selected for simulated reordering within the scope of its function (at most 1 in-flight access).

Once an access has been selected for reordering, it is checked along every other access until the end of the function scope. If an appropriate memory barrier is encountered, the access will no longer be considered for simulated reordering.

When the result of a memory operation should be ordered by a barrier, KCSAN can then detect data races where the conflict only occurs as a result of a missing barrier. Consider the example:

```
int x, flag;
void T1(void)
{
    x = 1;                  // data race!
    WRITE_ONCE(flag, 1);    // correct: smp_store_release(&flag, 1)
}
void T2(void)
```

```
{
    while (!READ_ONCE(flag));   // correct: smp_load_acquire(&flag)
    ... = x;                    // data race!
}
```

When weak memory modeling is enabled, KCSAN can consider `x` in `T1` for simulated reordering. After the write of `flag`, `x` is again checked for concurrent accesses: because `T2` is able to proceed after the write of `flag`, a data race is detected. With the correct barriers in place, `x` would not be considered for reordering after the proper release of `flag`, and no data race would be detected.

Deliberate trade-offs in complexity but also practical limitations mean only a subset of data races due to missing memory barriers can be detected. With currently available compiler support, the implementation is limited to modeling the effects of "buffering" (delaying accesses), since the runtime cannot "prefetch" accesses. Also recall that watchpoints are only set up for plain accesses, and the only access type for which KCSAN simulates reordering. This means reordering of marked accesses is not modeled.

A consequence of the above is that acquire operations do not require barrier instrumentation (no prefetching). Furthermore, marked accesses introducing address or control dependencies do not require special handling (the marked access cannot be reordered, later dependent accesses cannot be prefetched).

Key Properties

1. **Memory Overhead:** The overall memory overhead is only a few MiB depending on configuration. The current implementation uses a small array of longs to encode watchpoint information, which is negligible.
2. **Performance Overhead:** KCSAN's runtime aims to be minimal, using an efficient watchpoint encoding that does not require acquiring any shared locks in the fast-path. For kernel boot on a system with 8 CPUs:
    - 5.0x slow-down with the default KCSAN config;
    - 2.8x slow-down from runtime fast-path overhead only (set very large `KCSAN_SKIP_WATCH` and unset `KCSAN_SKIP_WATCH_RANDOMIZE`).
3. **Annotation Overheads:** Minimal annotations are required outside the KCSAN runtime. As a result, maintenance overheads are minimal as the kernel evolves.
4. **Detects Racy Writes from Devices:** Due to checking data values upon setting up watchpoints, racy writes from devices can also be detected.
5. **Memory Ordering:** KCSAN is aware of only a subset of LKMM ordering rules; this may result in missed data races (false negatives).
6. **Analysis Accuracy:** For observed executions, due to using a sampling strategy, the analysis is *unsound* (false negatives possible), but aims to be complete (no false positives).

Alternatives Considered

An alternative data race detection approach for the kernel can be found in the Kernel Thread Sanitizer (KTSAN). KTSAN is a happens-before data race detector, which explicitly establishes the happens-before order between memory operations, which can then be used to determine data races as defined in Data Races.

To build a correct happens-before relation, KTSAN must be aware of all ordering rules of the LKMM and synchronization primitives. Unfortunately, any omission leads to large numbers of false positives, which is especially detrimental in the context of the kernel which includes numerous custom synchronization mechanisms. To track the happens-before relation, KTSAN's implementation requires metadata for each memory location (shadow memory), which for each page corresponds to 4 pages of shadow memory, and can translate into overhead of tens of GiB on a large system.