Data-race detection in the Linux kernel

Marco Elver <elver@google.com>
Why do we need a race detector?

- Thinking about multiple threads of execution is notoriously difficult.
- Kernel's job inherently concurrent.
- Tension between performant vs. simpler synchronization mechanisms.
- Numerous advanced synchronization mechanisms.

Tool assistance!
The Kernel Concurrency Sanitizer (KCSAN)

- Kernel Concurrency Sanitizer (KCSAN): dynamic race detector (at runtime).
  - Detects "data races" by default (more with special assertions, discussed later).

- KCSAN merged into Linux 5.8.
  - Development version on Paul E. McKenney's -rcu tree.
Background
What are data races?

- C-language and compilers evolved oblivious to concurrency.
- Optimizing compilers are becoming more creative [1].
  - load tearing,
  - store tearing,
  - load fusing,
  - store fusing,
  - code reordering,
  - invented loads,
  - invented stores,
  - ... and more.

➢ Need to tell compiler about concurrent code.

What are data races?

"Data race" defined via language's *memory consistency model*.

- C-language and compilers no longer oblivious to concurrency:
  - C11 introduced memory model: "data races cause undefined behaviour" — not Linux's model!
- **Linux kernel** has its own *memory model (LKMM)*, giving semantics to concurrent code.
What are data races?

Data races (✘) occur if:

- Concurrent conflicting accesses;
  - they conflict if they access the same location and at least one is a write.
- At least one is a plain access (e.g. "x + 42").
  - vs. "marked" accesses: READ_ONCE(), WRITE_ONCE(), smp_load_acquire(), smp_store_release(), atomic_t, ...

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>✘ ... = x + 1;</td>
<td>✘ x = 0xf0f0;</td>
</tr>
<tr>
<td>✘ ... = x + 1;</td>
<td>WRITE_ONCE(x, 0xf0f0);</td>
</tr>
<tr>
<td>✘ ... = READ_ONCE(x) + 1;</td>
<td>✘ x = 0xf0f0;</td>
</tr>
</tbody>
</table>
| ✘ ... = READ_ONCE(x) + 1; | ✘ x++;
| ✘ x = 0xff00; | ✘ x = 0xff; |
| ✔ ... = READ_ONCE(x) + 1; | ✔ WRITE_ONCE(x, 0xf0f0); |
| ✔ WRITE_ONCE(x, 0xff00); | ✔ WRITE_ONCE(x, 0xff); |
What are data races?

*Data-race-free* code has several benefits:

1. **Well-defined.** Avoids having to reason about compiler and architecture to determine whether a given data race is benign.

2. **Fewer bugs.** Data races can also indicate higher-level race-condition bugs.
   - E.g. failing to synchronize accesses using spinlocks, mutexes, RCU, etc.

➤ Prevent bugs, and countless hours debugging elusive race conditions!
A day in the life of a compiler

```c
void foo(int *x)
{
    if (*x) a = 42;
    if (*x) b = 42;
}
```

optimize: fuse loads

```c
void foo(int *x)
{
    if (*x) {
        a = 42;
        b = 42;
    }
}
```
A day in the life of a compiler

```c
void foo(int *x)
{
    if (*x) a = 42;
    if (*x) b = 42;
}
```

```c
void foo(int *x)
{
    if (*x) {
        a = 42;
        b = 42;
    }
}
```

```c
void badwait(int *stop)
{
    while (!*stop);
}
```

```c
void badwait(int *stop)
{
    if (!*stop) {
        while(1);
    }
}
```
A day in the life of a compiler

```c
void badwait(int *stop)
{
    while (!*stop);
}
```

```c
void badwait(int *stop)
{
    if (!*stop)
    {
        while(1);
    }
}
```

```c
WRITE_ONCE(*stop, 1);
```
A day in the life of a compiler

```c
void badwait(int *stop) {
    while (!READ_ONCE(*stop));
}
```

```c
WRITE_ONCE(*stop, 1);
```

```c
void badwait(int *stop) {
    while (!*stop);
}
```

```c
void badwait(int *stop) {
    if (!*stop) {
        while(1);
    }
}
```
Data races often symptom of more serious issue

BUG: KCSAN: data-race in __fat_write_inode / fat12_ent_get

write to 0xfffff8881015f423c of 4 bytes by task 9966 on cpu 1:
  __fat_write_inode+0x246/0x510 fs/fat/inode.c:877
...

read to 0xfffff8881015f423d of 1 bytes by task 9960 on cpu 0:
  fat12_ent_get+0x5e/0x120 fs/fat/fatent.c:125
...

Careful, if symptom of higher-level issue!
Some numbers

Number of known fixes to address data races (since KCSAN was announced September 2019): ~60

Number of current KCSAN reports on syzbot: ~350

- Biggest challenge: filter and prioritize.

Want to also encourage testing to prevent issues:

```bash
$> git log --format=oneline v5.3..v5.8 |
grep -Ei '.*(fix|avoid) .*[-]race[-]'
wc -l
101
```
Data-race detection in the Linux kernel
Past attempts at data race detectors for the kernel

Kernel Thread Sanitizer (KTSAN) [1]: github.com/google/ktsan/wiki

- Detect data races at runtime.
- Compiler instrumentation.
- Runtime: Same algorithm as user space ThreadSanitizer (TSAN) v2.
  ○ `{gcc,clang}` -fsanitize=thread
- Happens-before race detector (vector clocks).

Pros:
- few false negatives, precise, detects memory ordering issues (missing memory barriers etc.).

Cons:
- scalability, memory overhead, false positives without annotating all synchronization primitives.

URL: github.com/google/ktsan/wiki#implementation
Past attempts at data race detectors for the kernel

Other interesting approaches:

- **RaceHound:** [github.com/kmrov/racehound](https://github.com/kmrov/racehound)
- **Based on DataCollider approach** [1]:
  - set HW breakpoint + delay;
  - if breakpoint triggered ⇒ race;
  - if value changed ⇒ race.

- ... probably more ...

**Why did they never make it into mainline?**

What is a reasonable design for the kernel?

<table>
<thead>
<tr>
<th>Requirement</th>
<th>RaceHound</th>
<th>DataCollider</th>
<th>Kernel Thread Sanitizer (KTSAN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime performance</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Low memory overhead</td>
<td>✔</td>
<td></td>
<td>✘</td>
</tr>
<tr>
<td>Prefer false negatives over false positives</td>
<td>✔</td>
<td></td>
<td>✘</td>
</tr>
<tr>
<td>Maintenance: unintrusive to rest of kernel</td>
<td>✔</td>
<td></td>
<td>✘</td>
</tr>
<tr>
<td>Scalable memory access instrumentation</td>
<td>✘</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Language-level access aware (LKMM-compatibility)</td>
<td>✘</td>
<td></td>
<td>✔</td>
</tr>
</tbody>
</table>
## What is a reasonable design for the kernel?

<table>
<thead>
<tr>
<th>Requirement</th>
<th>RaceHound</th>
<th>DataCollider</th>
<th>Kernel Thread Sanitizer (KTSAN)</th>
<th>Kernel Concurrency Sanitizer (KCSAN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime performance</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Low memory overhead</td>
<td>✔</td>
<td></td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>Prefer false negatives over false positives</td>
<td>✔</td>
<td></td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>Maintenance: unintrusive to rest of kernel</td>
<td>✔</td>
<td></td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>Scalable memory access instrumentation</td>
<td>✘</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Language-level access aware (LKMM-compatibility)</td>
<td>✘</td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
The Kernel Concurrency Sanitizer (KCSAN)

Dynamic data race detector (detecting races at runtime).

- **CONFIG_KCSAN=y**
  - Various other options to tweak behaviour.
  - x86-64; coming soon: ARM64.
  - Ports welcome: core code generic and portable.

- Need recent compiler: Clang 11 (Linux 5.8+), GCC 11 (Linux 5.9+)
  - Initially designed to work with most old compilers that have -fsanitize=thread, but had to change for merging into 5.8.

```python
BUG: KCSAN: data-race in <function-1> / <function-2>
<operation> to <address> of <size> bytes by <context-1> on cpu <nr>: <call trace from function-1>
...
(optional: locks held by context-1>
<operation> to <address> of <size> bytes by <context-2> on cpu <nr>: <call trace from function-2>
...
(optional: locks held by context-2>
```

Reported by Kernel Concurrency Sanitizer on:
```system info
```
KCSAN: Overview

*Basic idea:* Observe that 2 accesses happen concurrently.

➤ Catch races precisely when they happen!
KCSAN: Overview

*Which accesses*: let compiler instrument memory accesses.

```
int x;
...
x = 42;
...
... = x;
```

```
int x;
...
__tsan_write4(&x);
x = 42;
...
__tsan_read4(&x);
... = x;
```
KCSAN: Overview

- **Catch races using "soft" watchpoints:**
  - Set watchpoint, and stall access.
  - If watchpoint already exists ⇒ race.
  - If value changed ⇒ race.
  - Stall accesses with random delays to increase chance to observe race.
    - **Default:** uniform between [1,80] µs for tasks, [1,20] µs for interrupts.

- **Set watchpoints for all instrumented memory accesses.**
  - Uninstrumented accesses (plain or marked) will never result in false positives!

- **Sampling:** periodically set up watchpoints.
  - **Default:** every ~2000 accesses (uniform random [1,4000]).
  - **Caveat:** lower probability to detect infrequent races ⇒ offset by good stress tests, or fuzzers like **syzkaller**.
KCSAN: Runtime

(fast path)
KCSAN: Runtime

<table>
<thead>
<tr>
<th>find_watchpoint()</th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>✗</td>
<td>✔</td>
</tr>
<tr>
<td>write</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

process: enter runtime

check_access(ptr, size, type)

find_watchpoint(ptr, size, expect_write)

should_watch(ptr, size, type, ...)

setup_watchpoint(ptr, size, type)

delay, then check for race

was found -> value-change

no race

generate report

exit runtime
KCSAN: Runtime

(fast path)
Implementing the memory model

<table>
<thead>
<tr>
<th>Plain accesses</th>
<th>Should watch()</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔ (Sampling)</td>
<td></td>
</tr>
<tr>
<td>Marked / Atomic</td>
<td>✗</td>
</tr>
</tbody>
</table>
KCSAN: Runtime

(slow path)
KCSAN: Runtime

<table>
<thead>
<tr>
<th>find_watchpoint()</th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>write</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Implementing the memory model

<table>
<thead>
<tr>
<th>Implementing the memory model</th>
<th>should_watch()</th>
</tr>
</thead>
<tbody>
<tr>
<td>plain accesses</td>
<td>✔ (sampling)</td>
</tr>
<tr>
<td>marked / atomic</td>
<td>✘</td>
</tr>
</tbody>
</table>

```
KCSAN: Runtime

<table>
<thead>
<tr>
<th>find_watchpoint()</th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>✘</td>
<td>✔</td>
</tr>
<tr>
<td>write</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Implementing the memory model

<table>
<thead>
<tr>
<th>Implementing the memory model</th>
<th>should_watch()</th>
</tr>
</thead>
<tbody>
<tr>
<td>plain accesses</td>
<td>✔ (sampling)</td>
</tr>
<tr>
<td>marked / atomic</td>
<td>✘</td>
</tr>
</tbody>
</table>
```
KCSAN: Soft Watchpoints

- "Soft" watchpoints → flexible, portable, scales to arbitrary number.
- Array of longs (atomic_long_t).
- Indexed based on address page.
  - Can spill into adjacent slots.
  - Index also used to ensure matching producers/consumers for report metadata.
KCSAN: Soft Watchpoints

- Special encoding, to avoid multiple fields and lock-based synchronization.
- Enables use of atomic_long_t for access information.

Example: 64 bits per long, and 4 KiB pages
(Calculated based on PAGE_SIZE and BITS_PER_LONG)
Beyond data races
Concurrent bugs that are not data races

```c
spin_lock(&update.foo_lock);
/* Careful! There should be no other
writers to shared_foo! Readers ok. */
WRITE_ONCE(shared_foo, ...);
spin_unlock(&update.foo_lock);
```
Concurrency bugs that are not data races

Thread 0

spin_lock(&update_foo_lock);
/* Careful! There should be no other writers to shared_foo! Readers ok. */
WRITE_ONCE(shared_foo, ...);
spin_unlock(&update_foo_lock);

Thread 1

/* update_foo_lock does not need to be held! */
... = READ_ONCE(shared_foo);
Concurrent bugs that are not data races

Thread 0

```c
spin_lock(&update_foo_lock);
/* Careful! There should be no other writers to shared_foo! Readers ok. */
WRITE_ONCE(shared_foo, ...);
spin_unlock(&update_foo_lock);
```

Thread 1

```c
/* update_foo_lock does not need to be held! */
... = READ_ONCE(shared_foo);
```

Thread 2

```c
/* Bug! */
WRITE_ONCE(shared_foo, 42);
```
Concurrency bugs that are not data races

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
</table>
| spin_lock(&update_foo_lock); /* No other writers to shared_foo. */
| ASSERT_EXCLUSIVE_WRITER(shared_foo);
| WRITE_ONCE(shared_foo, ...);
| spin_unlock(&update_foo_lock); /* update_foo_lock does not need to be held! */
| ... = READ_ONCE(shared_foo); |
| /* Bug */ WRITE_ONCE(shared_foo, 42); |
How KCSAN can help find more bugs

- **ASSERT_EXCLUSIVE** family of macros.
  - Specify properties of concurrent code, where bugs are not normal data races.
  - Reported as: "BUG: KCSAN: assert: race in <func-1> / <func-2>"
  - Result of early discussion with community members who pointed out that data-race detection alone was not enough to check the complex concurrency designs found in the Linux kernel.

<table>
<thead>
<tr>
<th>Function</th>
<th>Concurrent writes</th>
<th>Concurrent reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSERT_EXCLUSIVEewriter(var)</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>ASSERT_EXCLUSIVEewriter_scoped(var)</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>ASSERT_EXCLUSIVEaccess(var)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>ASSERT_EXCLUSIVEaccess_scoped(var)</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>ASSERT_EXCLUSIVE_bits(var, mask)</td>
<td>~mask ✓</td>
<td>mask ✗</td>
</tr>
</tbody>
</table>
Conclusion
Early community feedback and iterate

Examples:

- `CONFIG_KCSAN_REPORT_VALUE_CHANGE_ONLY`
- `data_race(...)` macro
- `CONFIG_KCSAN_ASSUME.PLAIN_WRITES_ATOMIC`

```c
... = READ_ONCE(x) + 1;  x = 0xf0f0f0;
```

- `CONFIG_KCSAN_VERBOSE` (lockdep integration)
Open questions

A. How should we report data races from CI systems? (syzbot..)
   ○ Currently needs moderation, but also want expert eyes!
   ○ Keep sending one-by-one?
   ○ Or batch data race reports from subsystems?

B. How to deal with plain read-modify-writes ("++ , +=, -- , -=, |=, ...")?
   ○ Concurrent use is pervasive.
   ○ Some can safely be marked data_race().
   ○ But, in some cases really hard to say if safe. What to do?
Concurrency bugs should fear the big bad data-race detector

- Data races harmful: beware compiler, and/or symptom of deeper issues.
- Need tool assistance: growing kernel, many synchronization mechanisms.

The Kernel Concurrency Sanitizer (KCSAN)
- Available in mainline since Linux 5.8.
- Compile with: `CONFIG_KCSAN=y`


Links: github.com/google/ktsan/wiki/KCSAN
Backup: Some interesting KCSAN reports

KCSAN: data-race in _fat_write_inode / fat12_ent_get → fixed, invalid filesystem access
KCSAN: data-race in tun_get_user / tun_net_get_stats64 → fixed, tearing u64 on 32-bit arches
KCSAN: data-race in dyntick_save_progress_counter / rcu_irq_enter → fixed, non-atomic atomic_t access
KCSAN: data-race in sctp_assoc_migrate / sctp_hash_obj → fixed
KCSAN: data-race in gro_normal_list / napi_busy_loop → fixed
KCSAN: data-race in task_dump_owner / task_dump_owner → unfixed, potential security issue
KCSAN: data-race in generic_file_buffered_read / generic_file_buffered_read → unfixed, plain RMWs

More: syzkaller.appspot.com/upstream?manager=ci2-upstream-kcsan-gce