MENTORSHIP SERIES

LF Live MENTORSHIP SERIES

# The Linux Kernel Concurrency Sanitizer

Marco Elver, Senior Software Engineer, Google

## Introduction and Agenda

- Background on Data Races
- The Linux Kernel Memory Consistency Model
- Linux-Kernel Data-Race Detection with KCSAN
- Concurrency Bugs Beyond Data Races

## Scope and Expectations

- Understanding of Data Races
- Brief Introduction to Memory Consistency Models
- Building and Using Kernels with KCSAN

# Problem

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

**We need tool assistance!**

# Background

# What are data races?

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

⚠️ **Need to tell compiler about concurrent code**

📖 "Who's afraid of a big bad optimizing compiler?", LWN 2019. URL: [https://lwn.net/Articles/793253/](https://lwn.net/Articles/793253/)

# What are data races?

Defined via language's *memory consistency model* (more detail later):

- 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**, giving semantics to concurrent code

MENTORSHIP SERIES

# What are data races?

**Basic definition:**

1. *Concurrent conflicting accesses*
   - *They conflict if they access the same location and at least one is a write*
2. *At least one is a plain access (e.g. "x + 42")*
   - *Not specially marked for synchronization (compiler assumes no concurrency)*

# What are data races?

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

1. **Well-defined.** Avoids having to reason about compiler and architecture.
   – Avoid having to reason "Is this data race 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.
3. **Prevent bugs,** and countless hours debugging elusive race conditions!

# Concurrency vs. Compiler Optimizations

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

optimize: fuse loads

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

MENTORSHIP SERIES

# Concurrency vs. Compiler Optimizations

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

optimize: fuse loads

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

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

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

# Concurrency vs. Compiler Optimizations

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

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

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

# Concurrency vs. Compiler Optimizations

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

✔

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

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

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

MENTORSHIP SERIES

# Data races often symptom of more serious issue

```
BUG: KCSAN: data-race in __fat_write_inode / fat12_ent_get

write to 0xffff8881015f423c of 4 bytes by task 9966 on cpu 1:
 __fat_write_inode+0x246/0x510 fs/fat/inode.c:877
 ...

read to 0xffff8881015f423d of 1 bytes by task 9960 on cpu 0:
 fat12_ent_get+0x5e/0x120 fs/fat/fatent.c:125
 ...
```

**fat: don't allow to mount if the FAT length == 0**

```
If FAT length == 0, the image doesn't have any data. And it can be the
cause of overlapping the root dir and FAT entries.

Also Windows treats it as invalid format.

Reported-by: syzbot+6f1624f937d9d6911e2d@syzkaller.appspotmail.com
Signed-off-by: OGAWA Hirofumi <hirofumi@mail.parknet.co.jp>
Signed-off-by: Andrew Morton <akpm@linux-foundation.org>
Cc: Marco Elver <elver@google.com>
Cc: Dmitry Vyukov <dvyukov@google.com>
Link: http://lkml.kernel.org/r/87r1wz8mrd.fsf@mail.parknet.co.jp
Signed-off-by: Linus Torvalds <torvalds@linux-foundation.org>
```

**Diffstat**

| | | |
|---|---|---|
| -rw-r--r-- | fs/fat/inode.c | 6 |

1 files changed, 6 insertions, 0 deletions

```
diff --git a/fs/fat/inode.c b/fs/fat/inode.c
index e6e68b2274a5c..a0cf99debb1ec 100644
--- a/fs/fat/inode.c
+++ b/fs/fat/inode.c
@@ -1519,6 +1519,12 @@ static int fat_read_bpb(struct super_block *sb, struct
 		goto out;
 	}
 
+	if (bpb->fat_fat_length == 0 && bpb->fat32_length == 0) {
+		if (!silent)
+			fat_msg(sb, KERN_ERR, "bogus number of FAT sectors");
+		goto out;
+	}
+
 	error = 0;
```

**Careful, if symptom of higher-level issue!**

# The Linux Kernel Memory Consistency Model (LKMM)

# What is a memory consistency model?

- What's the behaviour of memory accesses on a multiprocessor system?
- Or simply: *What value does a read access observe?*
- To write correct concurrent code, programmer needs to understand the semantics of the system they are programming

**Memory consistency model specifies ordering guarantees of memory operations with which the programmer can reason about parallel programs**

# What is a memory consistency model?

**Exists at different levels in our stack:**

- At hardware level, architecture has a memory model (system-centric model)
  - x86-TSO, Armv8, Armv7, PowerPC, Alpha
- Programming language should have its own (programmer-centric model)
  - C++11 [1,2], C11, Java

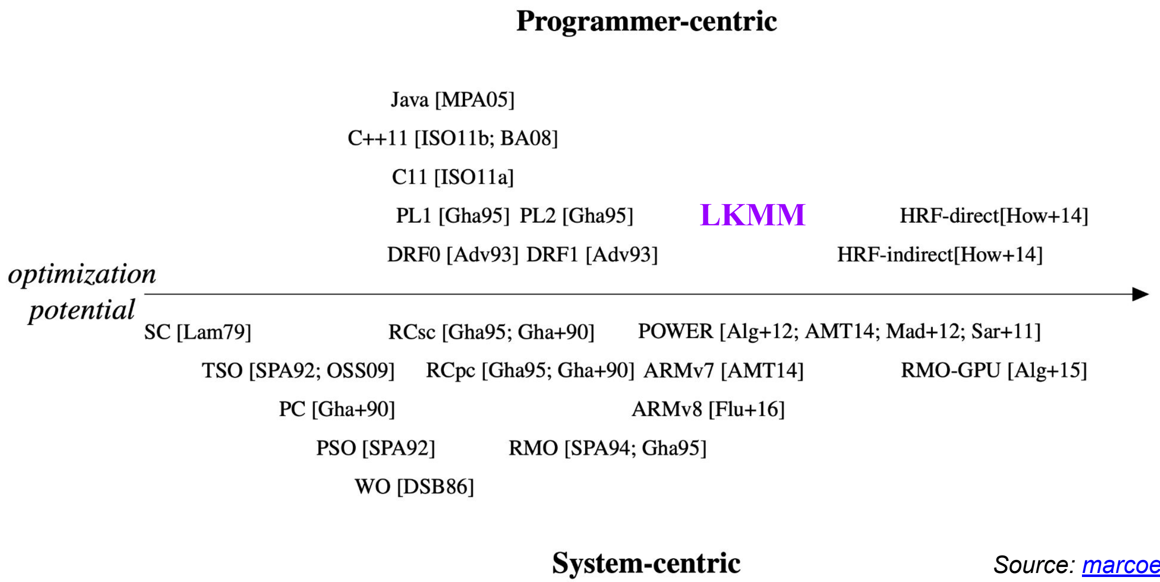
# Language-Level Memory Models

- Required to deal with compiler optimizations vs. concurrent code
- Distinguishes
  - "marked" (viz. atomic, or synchronization) and
  - "plain" (viz. "ordinary", non-atomic, or data) accesses
- Atomicity and ordering guarantees of marked accesses w.r.t. other accesses
- Marked atomic accesses building blocks for synchronization
- Compiler must not transform code in ways that would weaken memory model

📖 S. Adve, K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial", 1996

# Trade-offs Between Performance and Programmability

- Strictest and simplest model is [Sequential Consistency (SC)]
- Weak memory models offer more opportunities for speculation ⇒ Performance!




*Source:* [*marcoelver.com/res/melver-thesis.pdf*]

# The Linux Kernel Memory Consistency Model (LKMM)

- The Linux kernel's requirements resulted in a non-standard memory model
- Evolved, with many changes over the years (remember `ACCESS_ONCE()`?)
- The formal LKMM (tools/memory-model) incomplete vs. real code
- Informal documentation (memory-barriers.txt) not complete either: *"[...] This document is not a specification; it is intentionally (for the sake of brevity) and unintentionally (due to being human) incomplete. [...]"*

📖 J. Alglave et al., "Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel", 2018

## LKMM: Basic Marked Accesses

| Primitive | Result | Orders |
|---|---|---|
| `READ_ONCE(x)` | *read of x* | later dependent reads+marked writes |
| `WRITE_ONCE(x, Y)` | *write Y to x* | none |
| `smp_load_acquire(&x)` | *read of x* | later reads+writes |
| `smp_store_release(&x, Y)` | *write Y to x* | earlier reads+writes |
| `rcu_dereference(x)` | *read of x* | later dependent reads+marked writes |
| `smp_mb()` | *none* | earlier+later reads+writes |
| `smp_rmb()` | *none* | earlier+later reads |
| `smp_wmb()` | *none* | earlier+later writes |

## Dependency Ordering

- `READ_ONCE()` and `rcu_dereference()` orders later:
  - address-, data-, and control-dependent marked writes
  - address-dependent reads

# Dependency Ordering

```
/* Address dependency */
x = READ_ONCE(*foo);
… = READ_ONCE(*x);
```

```
/* Data dependency */
x = READ_ONCE(*foo);
x += 42;
WRITE_ONCE(*bar, x);
```

```
/* Control dependency */
x = READ_ONCE(*foo);
if (x) {
        WRITE_ONCE(*bar, 42);
}
```

```
/* Not ordered by control
dependency */
x = READ_ONCE(*foo);
if (x) {
        y = READ_ONCE(*bar);
}
```

**Warning:** Most tricky aspect of LKMM and likely to change in future because compilers can still break dependencies.

Further reading: [Status Report: Broken Dependency Orderings in the Linux Kernel]

# Many more marked accesses in LKMM

- All atomic_t accessors
- Atomic read-modify-writes: xchg(), cmpxchg() and variants
- Atomic bitops

# What are data races in the LKMM?

**Data races (✘) occur if:**

- <u>Concurrent</u> conflicting accesses
  - they conflict if they access the <u>same location</u> and <u>at least one is a write</u>
- At least one is a <u>plain</u> access

| | Thread 0 | Thread 1 |
|---|---|---|
| ✘ | `... = x + 1;` | `x = 0xf0f0;` |
| ✘ | `... = x + 1;` | `WRITE_ONCE(x, 0xf0f0);` |
| ✘ | `... = READ_ONCE(x) + 1;` | `x = 0xf0f0;` |
| ✘ | `... = 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);` |

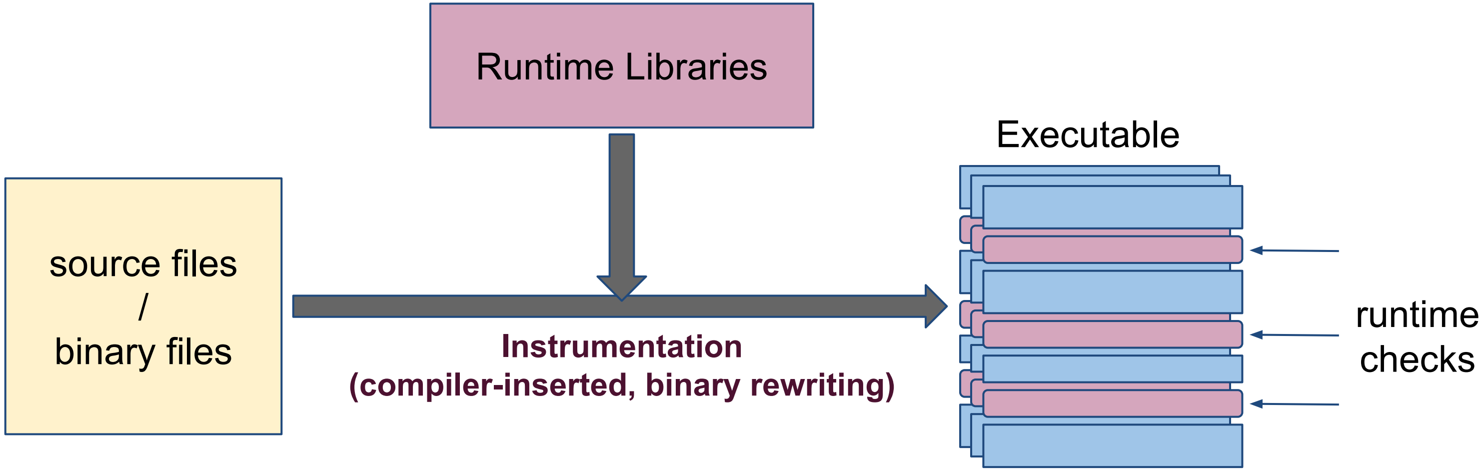
## Intentional Data Races

- The Linux kernel says that data races do not result in undefined behaviour of the whole kernel
- Locally "undefined" behaviour: where code still operates correctly even with potentially random data, data races are tolerated (truly "benign" data races)
- Mark such data races with "`data_race(.. data-racy expression ..)`"
    - Helps tooling understand they are intentional
    - Document intent

**For more guidance, see access-marking.txt**

# Linux-Kernel Data-Race Detection

# Dynamic Analysis



## Past Attempts at Kernel Data-Race Detectors

**Kernel Thread Sanitizer (KTSAN)**: google.github.io/kernel-sanitizers/KTSAN.html

- Compiler-instrumentation based (`-fsanitize=thread`)
- **Runtime:** Same algorithm as user space ThreadSanitizer (TSan v2)
    - 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

# Past Attempts at Kernel Data-Race Detectors

```
int x;
…
x = 42;
…
… = x;
```

`-fsanitize=thread`

```
int x;
…
__tsan_write4(&x);
x = 42;
…
__tsan_read4(&x);
… = x;
```

# Past Attempts at Kernel Data-Race Detectors

**Watchpoint-based race detection:**

- **RaceHound:** github.com/kmrov/racehound
- **DataCollider:** John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. "Effective Data-Race Detection for the Kernel", OSDI 2010

- **Basic Idea:**
  - set HW watchpoint + delay
  - if breakpoint triggered ⇒ race!
  - if value changed ⇒ race!

**Why did they never make it into the mainline Linux kernel?**

# Unique Linux-Kernel Requirements

| *Requirement* | *RaceHound* | *DataCollider* | *Kernel Thread Sanitizer (KTSAN)* | *Kernel Concurrency Sanitizer (KCSAN)* |
|---|---|---|---|---|
| Runtime performance | ✔ | | ✔ | ✔ |
| Low memory overhead | ✔ | | ✘ | ✔ |
| Prefer false negatives over false positives | ✔ | | ✘ | ✔ |
| Maintenance: unintrusive to rest of kernel | ✔ | | ✘ | ✔ |
| Scalable memory access instrumentation | ✘ | ✔ | ✔ | ✔ |
| Language-level access aware (LKMM-compatibility) | ✘ | | ✔ | ✔ |

# The Kernel Concurrency Sanitizer (KCSAN)

- Compiler-instrumentation based dynamic race detector
- Detects "data races" by default (more with special assertions, discussed later)
- Available since Linux 5.8
- Notable improvements (and relevant release):
  - Distinguishing read-modify-write accesses (5.10)
  - Show value changes (5.14)
  - More filtering data races (5.15)
  - Detection of data races due to missing memory barriers (5.17)

LF Live MENTORSHIP SERIES

# The Kernel Concurrency Sanitizer (KCSAN)

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

**Which accesses:** Let compiler instrument memory accesses

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

-fsanitize=thread →

```c
int x;
...
__tsan_write4(&x);
x = 42;
...
__tsan_read4(&x);
... = x;
```

# The Kernel Concurrency Sanitizer (KCSAN)

- 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

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

# The Kernel Concurrency Sanitizer (KCSAN)

**Usage:**

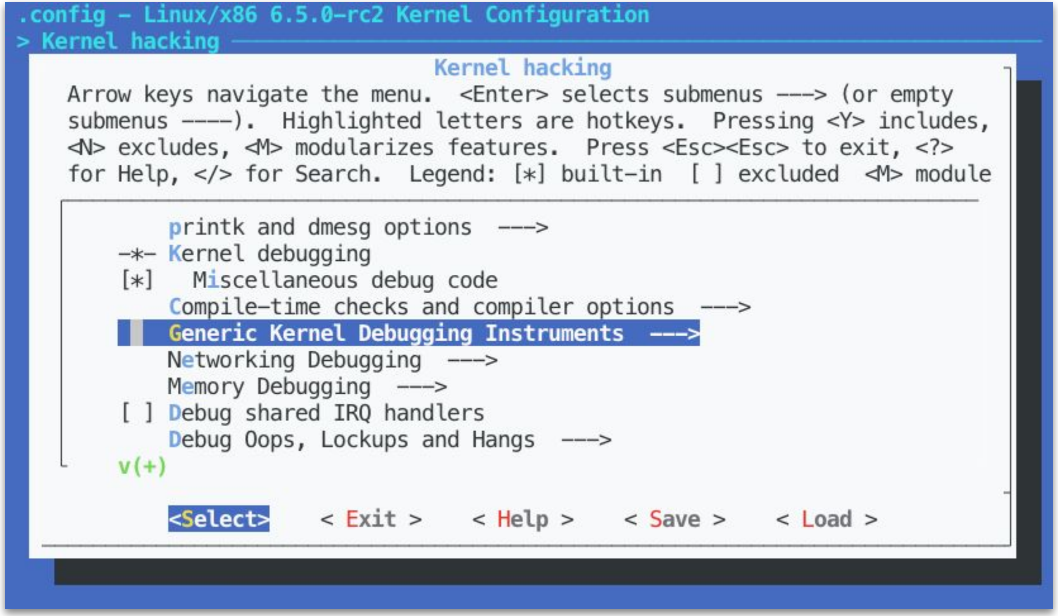
- **Supported architectures:** x86-64, arm64, s390, mips, powerp, xtensa
- **Compiler requirement:** Clang 11+, GCC 11+
- Build your kernel with **`CONFIG_KCSAN=y`**
  - For debugging and testing kernel
  - **Not** recommended for production kernels – more than 5x slowdown
- **Suggested:** `CONFIG_KCSAN_STRICT=y` (since 5.17)
  - "Strict" LKMM rules (but as of 6.4 still noisy)
  - Includes weak memory modeling (detect missing memory barriers)

# make menuconfig

```
.config - Linux/x86 6.5.0-rc2 Kernel Configuration
                    Linux/x86 6.5.0-rc2 Kernel Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y> includes,
  <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?>
  for Help, </> for Search.  Legend: [*] built-in  [ ] excluded  <M> module
 ----^(-)--------------------------------------------------------------------
         Executable file formats  --->
         Memory Management options  --->
     [*] Networking support  --->
         Device Drivers  --->
         File systems  --->
         Security options  --->
     -*- Cryptographic API  --->
         Library routines  --->
         Kernel hacking  --->

       <Select>    < Exit >    < Help >    < Save >    < Load >
```

# make menuconfig



# make menuconfig

```
.config - Linux/x86 6.5.0-rc2 Kernel Configuration
> Kernel hacking > Generic Kernel Debugging Instruments
                    Generic Kernel Debugging Instruments
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y> includes,
  <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?>
  for Help, </> for Search.  Legend: [*] built-in  [ ] excluded  <M> module

      [*] Magic SysRq key
      (0x1) Enable magic SysRq key functions by default
      [*]   Enable magic SysRq key over serial
      ()      Char sequence that enables magic SysRq over serial
      -*- Debug Filesystem
            Debugfs default access (Access normal)  --->
      [ ] KGDB: kernel debugger  ----
      [ ] Undefined behaviour sanity checker  ----
      [*] KCSAN: dynamic data race detector  --->

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

# make menuconfig

```
.config - Linux/x86 6.5.0-rc2 Kernel Configuration
[...] > Generic Kernel Debugging Instruments > KCSAN: dynamic data race detector
                        KCSAN: dynamic data race detector
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y> includes,
  <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?>
  for Help, </> for Search.  Legend: [*] built-in  [ ] excluded  <M> module

      --- KCSAN: dynamic data race detector
      [*]    Perform short selftests on boot
      <M>    KCSAN test for integrated runtime behaviour
      [*]    Early enable during boot
      (64)   Number of available watchpoints
      (80)   Delay in microseconds (for tasks)
      (20)   Delay in microseconds (for interrupts)
      [*]    Randomize above delays
      (4000) Skip instructions before setting up watchpoint
      v(+)

          <Select>    < Exit >    < Help >    < Save >    < Load >
```

# make menuconfig

```
.config - Linux/x86 6.5.0-rc2 Kernel Configuration
[...] > Generic Kernel Debugging Instruments > KCSAN: dynamic data race detector
                        KCSAN: dynamic data race detector
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y> includes,
  <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?>
  for Help, </> for Search.  Legend: [*] built-in  [ ] excluded  <M> module
      ^(-)
      (80)   Delay in microseconds (for tasks)
      (20)   Delay in microseconds (for interrupts)
      [*]    Randomize above delays
      (4000) Skip instructions before setting up watchpoint
      [*]    Randomize watchpoint instruction skip count
      (3000) Duration in milliseconds, in which any given race is only rep
      [*]    Report races of unknown origin
      [*]    Strict data-race checking
      [*]      Enable weak memory modeling to detect missing memory barrier
      v(+)

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

# Booting the Kernel with KCSAN

```
[    0.653289] Freeing SMP alternatives memory: 52K
[    0.653823] pid_max: default: 32768 minimum: 301
[    0.654967] LSM: initializing lsm=capability,selinux,integrity
[    0.655855] SELinux:  Initializing.
[    0.657059] Mount-cache hash table entries: 32768 (order: 6, 262144 bytes, linea
r)
[    0.657841] Mountpoint-cache hash table entries: 32768 (order: 6, 262144 bytes,
linear)
[    0.660704] kcsan: enabled early
[    0.660823] kcsan: non-strict mode configured - use CONFIG_KCSAN_STRICT=y to see
 all data races
[    0.662170] smpboot: CPU0: Intel(R) Xeon(R) Gold 6154 CPU @ 3.00GHz (family: 0x6
, model: 0x55, stepping: 0x4)
[    0.664104] RCU Tasks: Setting shift to 3 and lim to 1 rcu_task_cb_adjust=1.
[    0.664900] Performance Events: Skylake events, full-width counters, Intel PMU d
```

# Understanding KCSAN Reports

- **Title:** Shows which 2 functions raced
- **Access information:** Shows type of operation (read, write, read-write, and if marked), address, size, which task (or interrupt), and on which cpu
- **Optional:** Shows *lockdep* info if compiled with:
  - `CONFIG_PROVE_LOCKING=y`
  - `CONFIG_KCSAN_VERBOSE=y`

title / summary

```
==================================================================
BUG: KCSAN: data-race in <function-1> / <function-2>
```

access 1

```
<operation> to <address> of <size> bytes by <context-1> on cpu <nr>:
 <call trace from function-1>
 ...
```

lockdep info (optional)

```
<optional: locks held by context-1>
```

access 2

```
<operation> to <address> of <size> bytes by <context-2> on cpu <nr>:
 <call trace from function-2>
 ...
```

lockdep info (optional)

```
<optional: locks held by context-2>

Reported by Kernel Concurrency Sanitizer on:
<system info>
==================================================================
```

# Severity of Data Races

*Often the existence of a data race is merely a symptom of a bigger issue!*

**Data race may point out the following concurrency bugs:**

A. **Race-condition bugs** where the resulting **error manifests as a data race,** followed by eventual **system failure**. Simply marking the accesses does not fix the problem. The fix requires more invasive changes to program logic (for example adding required locking).
B. **Miscompilation** may introduce bugs that can lead to **system failure**. The fix requires using *appropriate marked atomic accesses*. Requires no fundamental changes in program logic to fix.
C. **Miscompilation** may introduce tolerated **inaccuracies** *("benign data race")*, but does not lead to system failure. Typically approximate diagnostics. Marking with `data_race(..)` is sufficient.

**Important: Avoid hiding bugs of type (A) by blindly marking data races!**

# Beyond Data Races

# 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);
```

# Concurrency Bugs That Are Not Data Races

| Thread 0 | Thread 1 |
| --- | --- |
| `spin_lock(&update_foo_lock);`<br>`/* Careful! There should be no other writers to shared_foo! Readers ok. */`<br>`WRITE_ONCE(shared_foo, ...);`<br>`spin_unlock(&update_foo_lock);` | `/* update_foo_lock does not need to be held! */`<br>`... = READ_ONCE(shared_foo);` |

# Concurrency Bugs That Are Not Data Races

| Thread 0 | Thread 1 | Thread 2 |
|---|---|---|
| `spin_lock(&update_foo_lock);`<br>`/* Careful! There should be no other writers to shared_foo! Readers ok. */`<br>`WRITE_ONCE(shared_foo, ...);`<br>`spin_unlock(&update_foo_lock);` | `/* update_foo_lock does not need to be held! */`<br>`... = READ_ONCE(shared_foo);` | `/* Bug! */`<br>`WRITE_ONCE(shared_foo, 42);` |

# Concurrency Bugs That Are Not Data Races

| Thread 0 | Thread 1 | Thread 2 |
| --- | --- | --- |
| `spin_lock(&update_foo_lock);`<br>`/* No other writers to shared_foo. */`<br>`ASSERT_EXCLUSIVE_WRITER(shared_foo);`<br>`WRITE_ONCE(shared_foo, ...);`<br>`spin_unlock(&update_foo_lock);` | `/* update_foo_lock does not`<br>`need to be held! */`<br>`... = READ_ONCE(shared_foo);` | ~~/* Bug! */~~<br>~~WRITE_ONCE(shared_foo, 42);~~ |

# Detecting More Concurrency 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 <func1> / <func2>`

| | concurrent writes | concurrent reads |
|---|---|---|
| **`ASSERT_EXCLUSIVE_WRITER`**(*var*)<br>**`ASSERT_EXCLUSIVE_WRITER_SCOPED`**(*var*) | ✘ | ✔ |
| **`ASSERT_EXCLUSIVE_ACCESS`**(*var*)<br>**`ASSERT_EXCLUSIVE_ACCESS_SCOPED`**(*var*) | ✘ | ✘ |
| **`ASSERT_EXCLUSIVE_BITS`**(*var*, *mask*) | `~mask` ✔ `mask` ✘ | ✔ |

## Concurrency Testing Best Practices

1. Design test cases to cover both expected and unexpected interleavings
2. Ensure to include test cases that cover different concurrency aspects of the code
3. Ensure to include test cases that mimic real-world scenarios
4. Stress test with a high number of threads that simulates worst case scenarios
5. Design test cases that quickly execute to-be-tested code repeatedly

🎥 Brendan Higgins, "KUnit Testing Strategies", LF Webinar 2021

🎥 Andrey Konovalov, "Fuzzing the Linux Kernel", LF Webinar 2021

# Summary

# Summary

- Concurrency in the Linux kernel is challenging
- The LKMM provides the foundation for writing concurrent code in the kernel
- Use KCSAN to help detect concurrency bugs early, and avoid data races

**Kernel Documentation: [docs.kernel.org/dev-tools/kcsan.html](docs.kernel.org/dev-tools/kcsan.html)**

Marco Elver, Paul E. McKenney, Dmitry Vyukov, Andrey Konovalov, Alexander Potapenko, Kostya Serebryany, Alan Stern, Andrea Parri, Akira Yokosawa, Peter Zijlstra, Will Deacon, Daniel Lustig, Boqun Feng, Joel Fernandes, Jade Alglave, and Luc Maranget. "**Concurrency bugs should fear the big bad data-race detector.**" Linux Weekly News (LWN), 2020. URL: https://lwn.net/Articles/816850/

LF live MENTORSHIP SERIES

# Thank you for joining us today!

We hope it will be helpful in your journey to learning more about effective and productive participation in open source projects. We will leave you with a few additional resources for your continued learning:

- The LF Mentoring Program is designed to help new developers with necessary skills and resources to experiment, learn and contribute effectively to open source communities.
- Outreachy remote internships program supports diversity in open source and free software
- Linux Foundation Training offers a wide range of free courses, webinars, tutorials and publications to help you explore the open source technology landscape.
- Linux Foundation Events also provide educational content across a range of skill levels and topics, as well as the chance to meet others in the community, to collaborate, exchange ideas, expand job opportunities and more. You can find all events at events.linuxfoundation.org.