Data-race detection in the Linux kernel

Marco Elver <elver@google.com>

LINUX
PLUMBERS CONFERENCE / August 24-28 2020

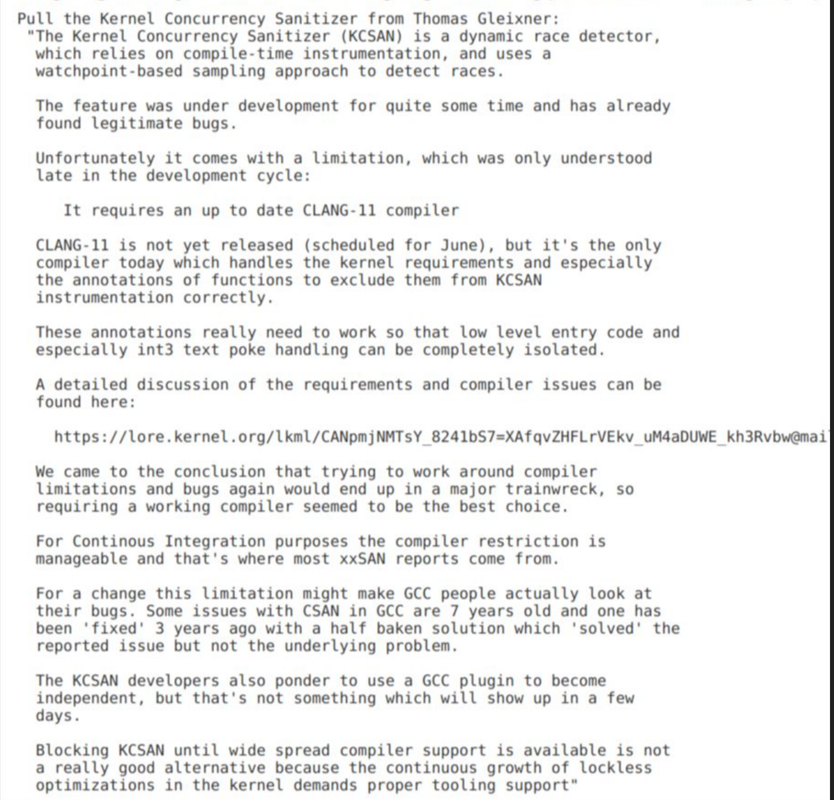
Why do we need a race detector?

- 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.



Merge tag 'locking-kcsan-2020-06-11' of git://git.kernel.org/pub/scm/linux/kernel/git/tip/tip

```
Pull the Kernel Concurrency Sanitizer from Thomas Gleixner:
 "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.

  The feature was under development for quite some time and has already
  found legitimate bugs.

  Unfortunately it comes with a limitation, which was only understood
  late in the development cycle:

     It requires an up to date CLANG-11 compiler

  CLANG-11 is not yet released (scheduled for June), but it's the only
  compiler today which handles the kernel requirements and especially
  the annotations of functions to exclude them from KCSAN
  instrumentation correctly.

  These annotations really need to work so that low level entry code and
  especially int3 text poke handling can be completely isolated.

  A detailed discussion of the requirements and compiler issues can be
  found here:

    https://lore.kernel.org/lkml/CANpmjNMTsY_8241bS7=XAfqvZHFLrVEkv_uM4aDUWE_kh3Rvbw@mai

  We came to the conclusion that trying to work around compiler
  limitations and bugs again would end up in a major trainwreck, so
  requiring a working compiler seemed to be the best choice.

  For Continous Integration purposes the compiler restriction is
  manageable and that's where most xxSAN reports come from.

  For a change this limitation might make GCC people actually look at
  their bugs. Some issues with CSAN in GCC are 7 years old and one has
  been 'fixed' 3 years ago with a half baken solution which 'solved' the
  reported issue but not the underlying problem.

  The KCSAN developers also ponder to use a GCC plugin to become
  independent, but that's not something which will show up in a few
  days.

  Blocking KCSAN until wide spread compiler support is available is not
  a really good alternative because the continuous growth of lockless
  optimizations in the kernel demands proper tooling support"
```

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.

[1] Jade Alglave, Will Deacon, Boqun Feng, David Howells, Daniel Lustig, Luc Maranget, Paul E. McKenney, Andrea Parri, Nicholas Piggin, Alan Stern, Akira Yokosawa, and Peter Zijlstra. "Who's afraid of a big bad optimizing compiler?", LWN 2019. URL: https://lwn.net/Articles/793253/

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`, …

	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);`

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!**

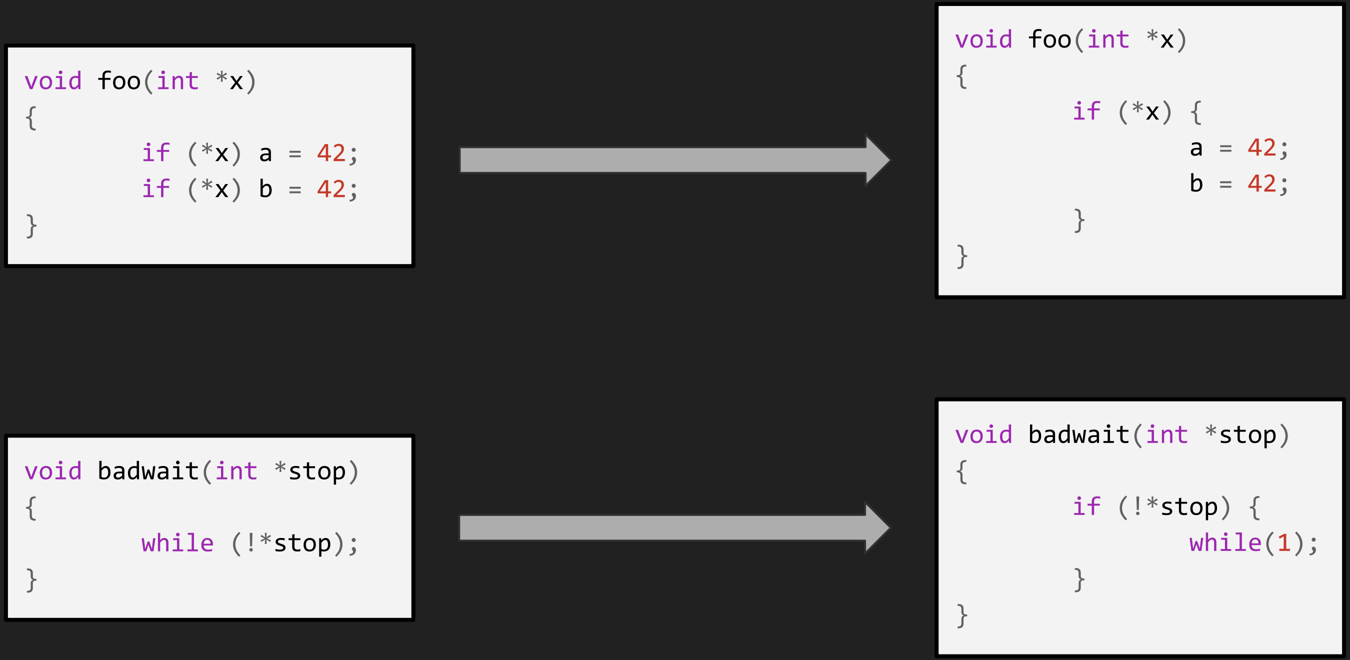
A day in the life of a compiler

```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



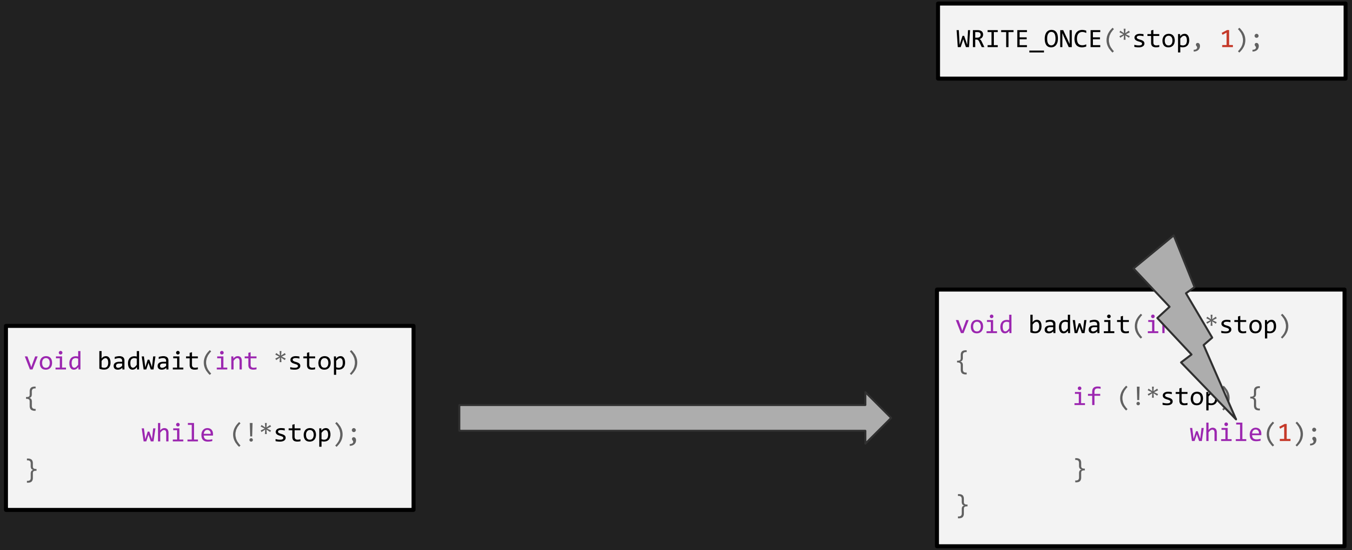
```c
void foo(int *x)
{
        if (*x) a = 42;
        if (*x) b = 42;
}
```

```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

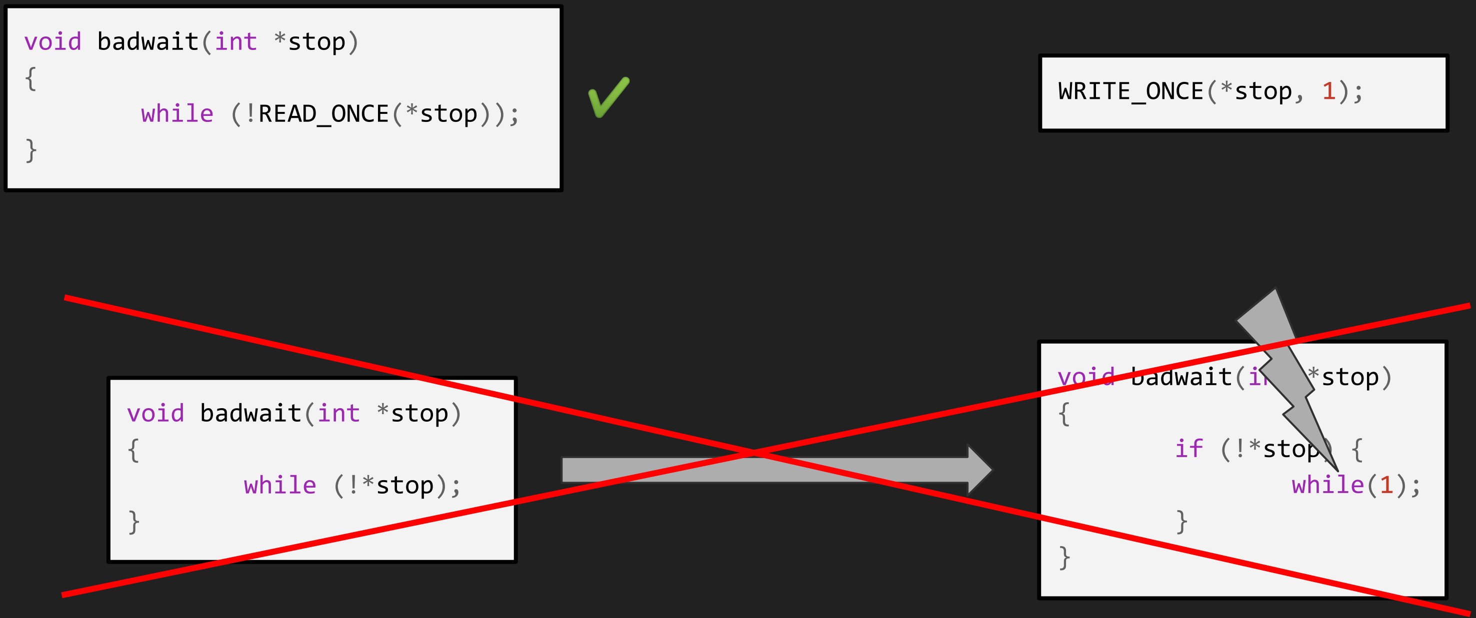


```
WRITE_ONCE(*stop, 1);
```

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

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

A day in the life of a compiler



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

✓

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

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

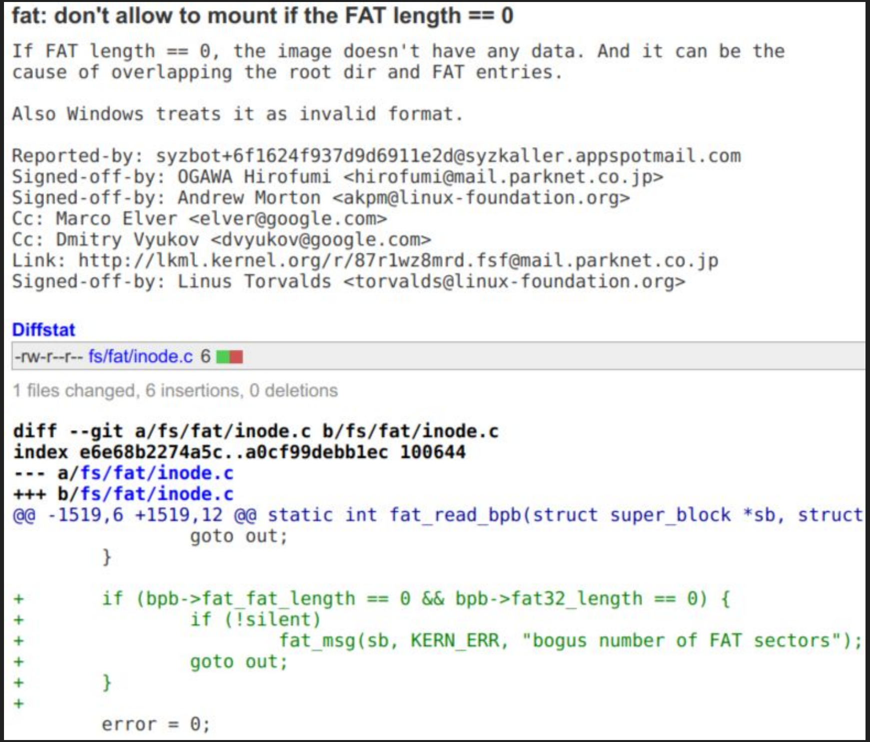
```
void badwait(i[illegible] *stop)
{
        if (!*stop) {
                while(1);
        }
}
```

Data races often symptom of more serious issue

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

write to 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..a0cf99debblec 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!

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:

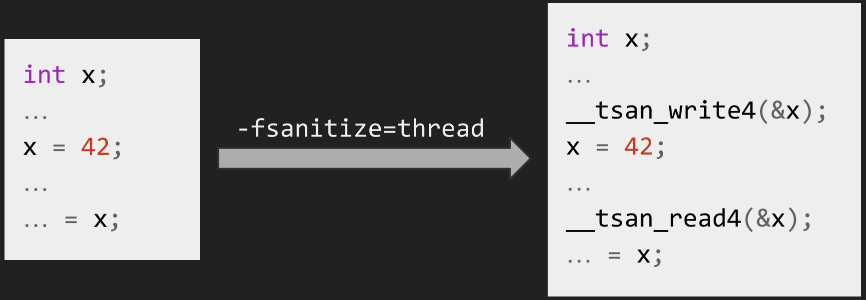
```
$> git log --format=oneline v5.3..v5.8 |
      grep -Ei '(fix|avoid) .*[ -]race[ -]' |
      wc -l
101
```

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).



```
int x;
…
x = 42;
…
… = x;
```

-fsanitize=thread

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

Pros:

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

Cons:

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

[1] Andrey Konovalov. "KernelThreadSanitizer (KTSAN): a data race detector for the Linux kernel", 2015.
URL: github.com/google/ktsan/wiki#implementation

Past attempts at data race detectors for the kernel

Other interesting approaches:

- RaceHound: 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?

---

[1] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. "Effective Data-Race Detection for the Kernel", OSDI 2010. URL: https://www.usenix.org/legacy/events/osdi10/tech/full_papers/Erickson.pdf

What is a reasonable design for the kernel?

<table>
<tr><th>Requirement</th><th>RaceHound</th><th>DataCollider</th><th>Kernel Thread Sanitizer (KTSAN)</th></tr>
<tr><td>Runtime performance</td><td colspan="2">✓</td><td>✓</td></tr>
<tr><td>Low memory overhead</td><td colspan="2">✓</td><td>✘</td></tr>
<tr><td>Prefer false negatives over false positives</td><td colspan="2">✓</td><td>✘</td></tr>
<tr><td>Maintenance: unintrusive to rest of kernel</td><td colspan="2">✓</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 colspan="2">✘</td><td>✓</td></tr>
</table>

What is a reasonable design for the kernel?

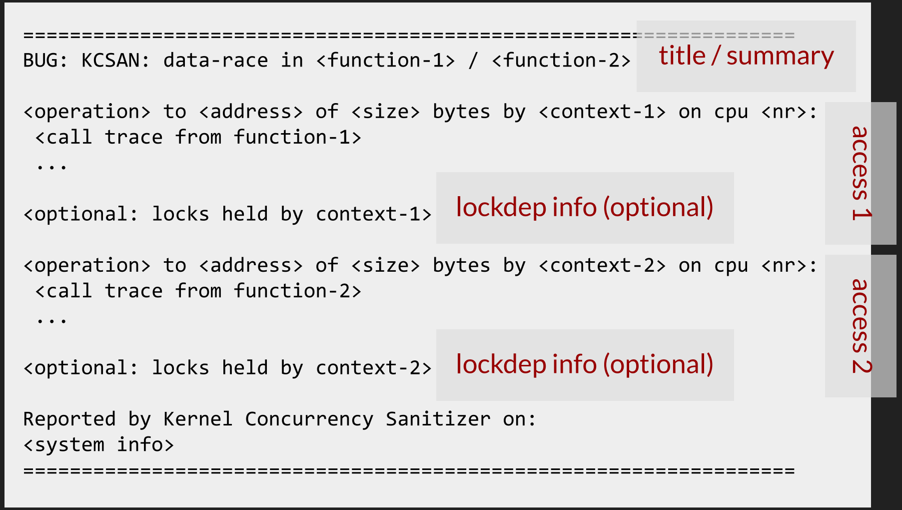
<table>
<tr><th>Requirement</th><th>RaceHound</th><th>DataCollider</th><th>Kernel Thread Sanitizer (KTSAN)</th><th>Kernel Concurrency Sanitizer (KCSAN)</th></tr>
<tr><td>Runtime performance</td><td colspan="2">✓</td><td>✓</td><td>✓</td></tr>
<tr><td>Low memory overhead</td><td colspan="2">✓</td><td>✘</td><td>✓</td></tr>
<tr><td>Prefer false negatives over false positives</td><td colspan="2">✓</td><td>✘</td><td>✓</td></tr>
<tr><td>Maintenance: unintrusive to rest of kernel</td><td colspan="2">✓</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 colspan="2">✘</td><td>✓</td><td>✓</td></tr>
</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.



```
==================================================================
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>
==================================================================
```

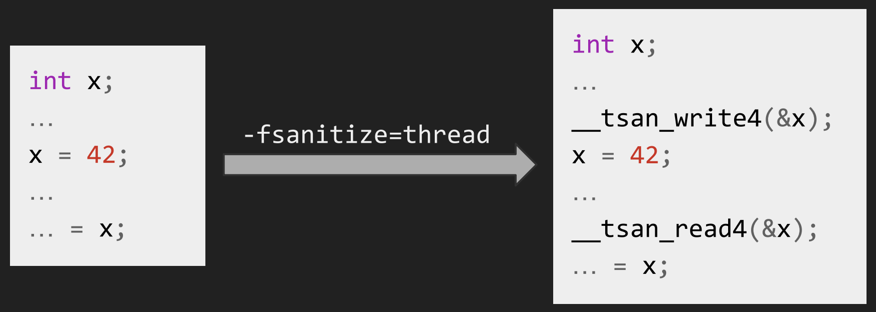
KCSAN: Overview

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;
```

`-fsanitize=thread`

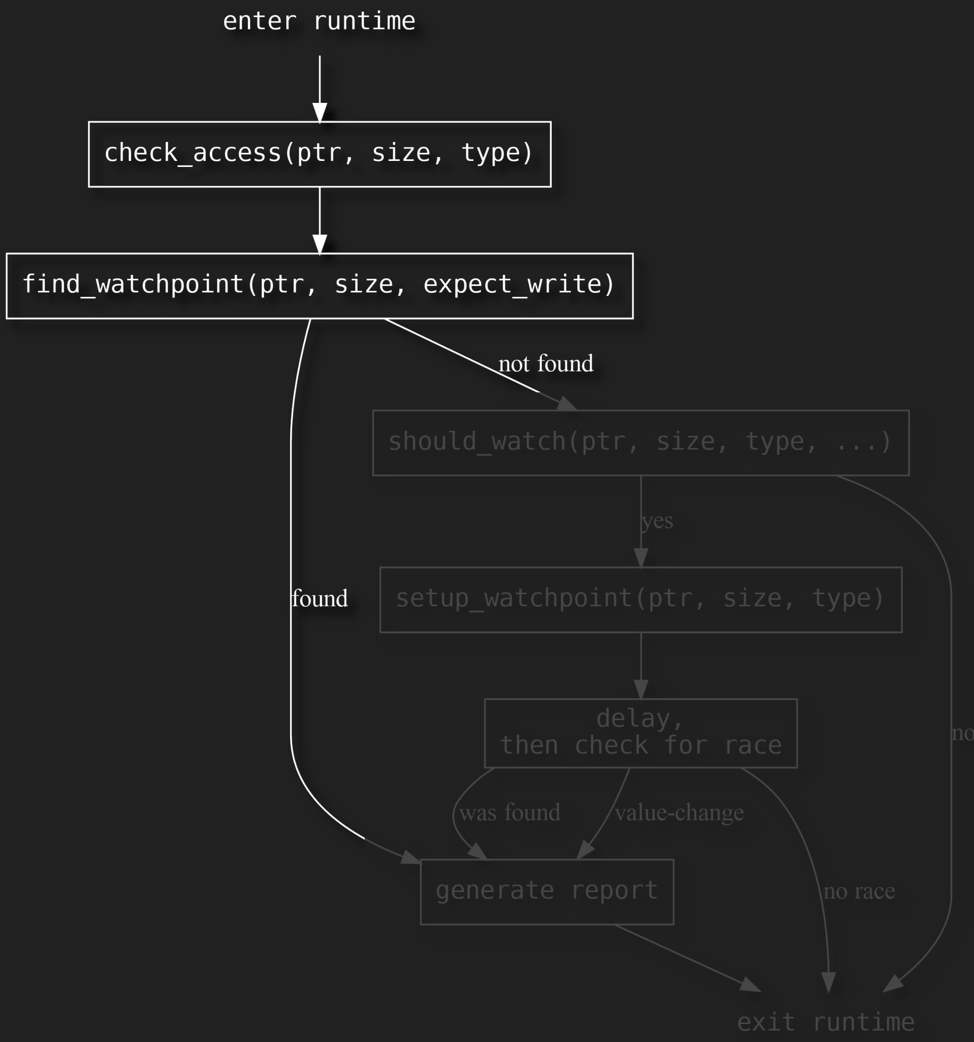
```
int x;
…
__tsan_write4(&x);
x = 42;
…
__tsan_read4(&x);
… = x;
```

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.

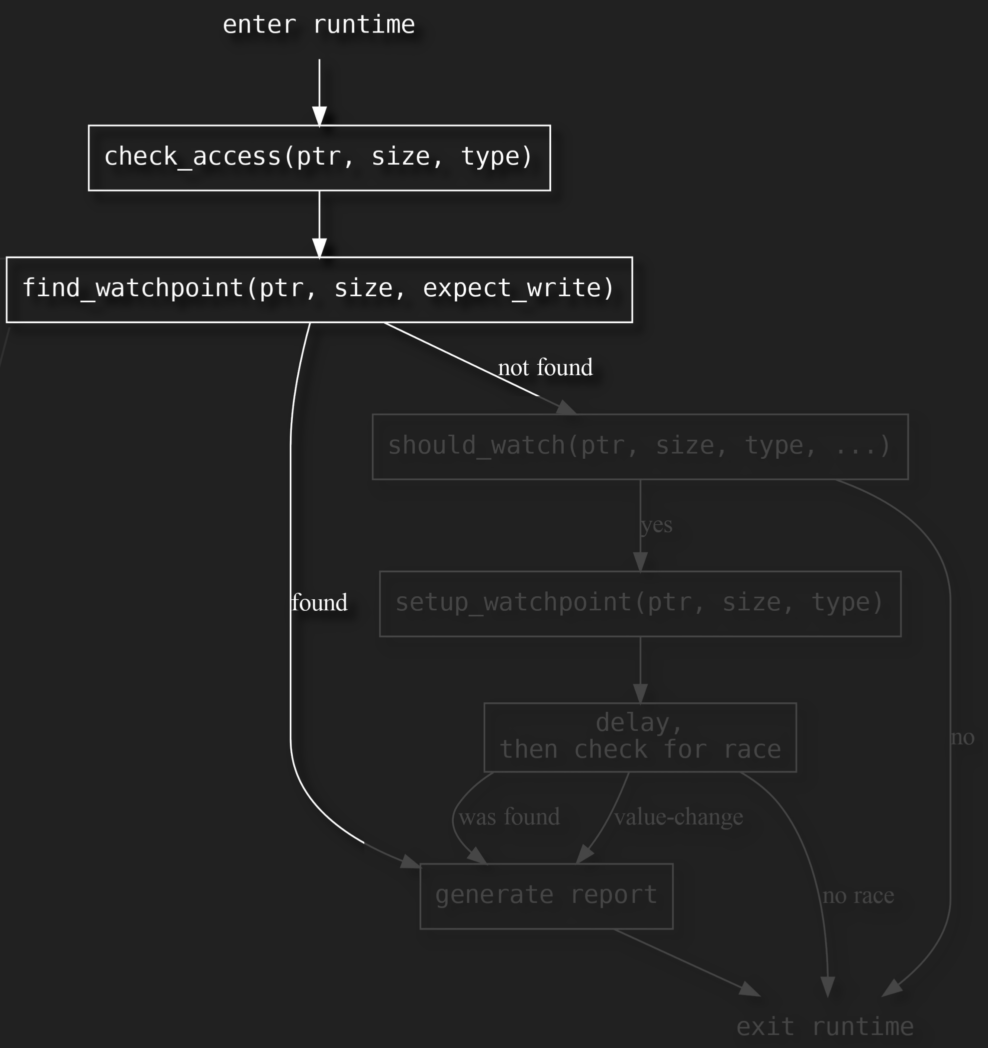
KCSAN: Runtime

(fast path)



KCSAN: Runtime

find_watchpoint()	read	write
read	✗	✓
write	✓	✓



enter runtime

check_access(ptr, size, type)

find_watchpoint(ptr, size, expect_write)

not found

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

yes

found

setup_watchpoint(ptr, size, type)

no

delay,
then check for race

was found

value-change

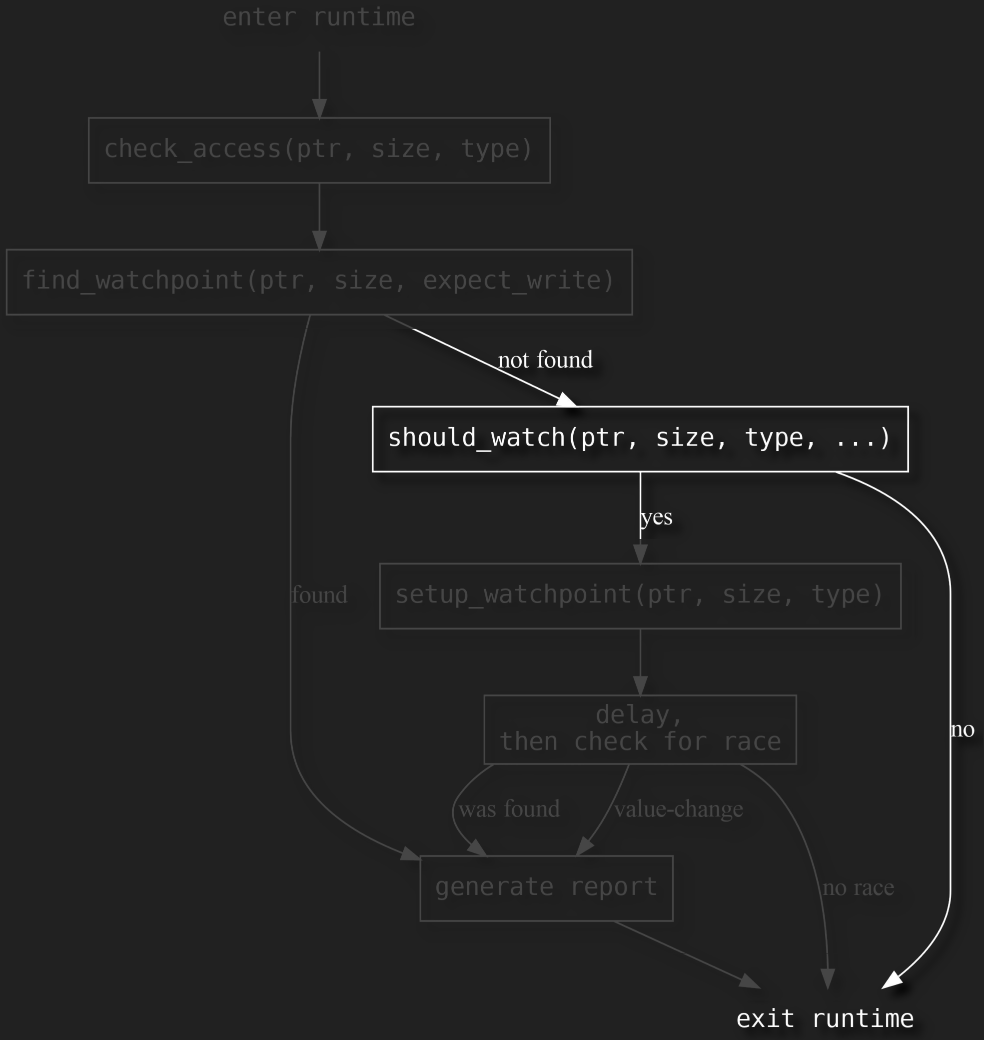
generate report

no race

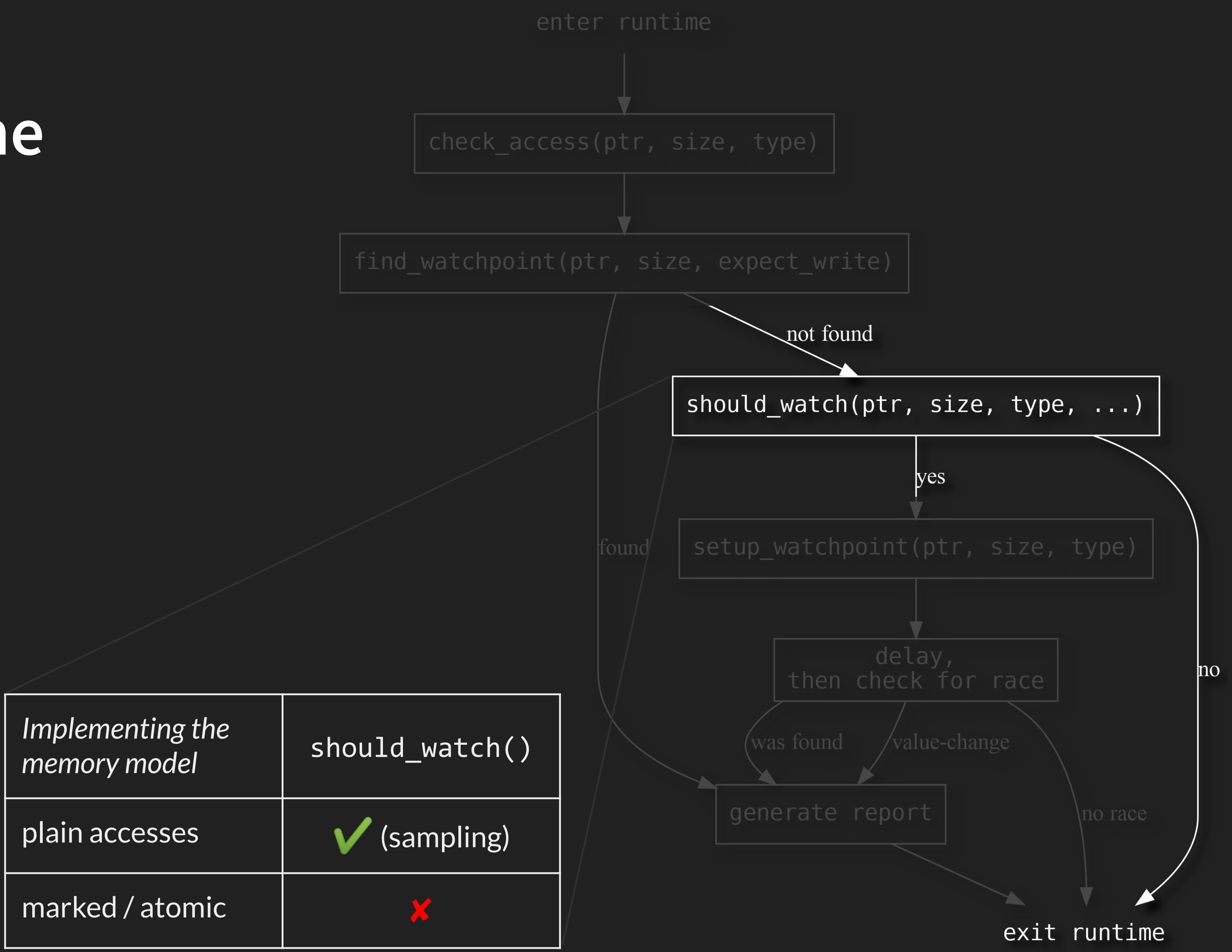
exit runtime

KCSAN: Runtime

(fast path)



KCSAN: Runtime



enter runtime

check_access(ptr, size, type)

find_watchpoint(ptr, size, expect_write)

not found

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

yes

setup_watchpoint(ptr, size, type)

delay,
then check for race

found

was found

value-change

no race

no

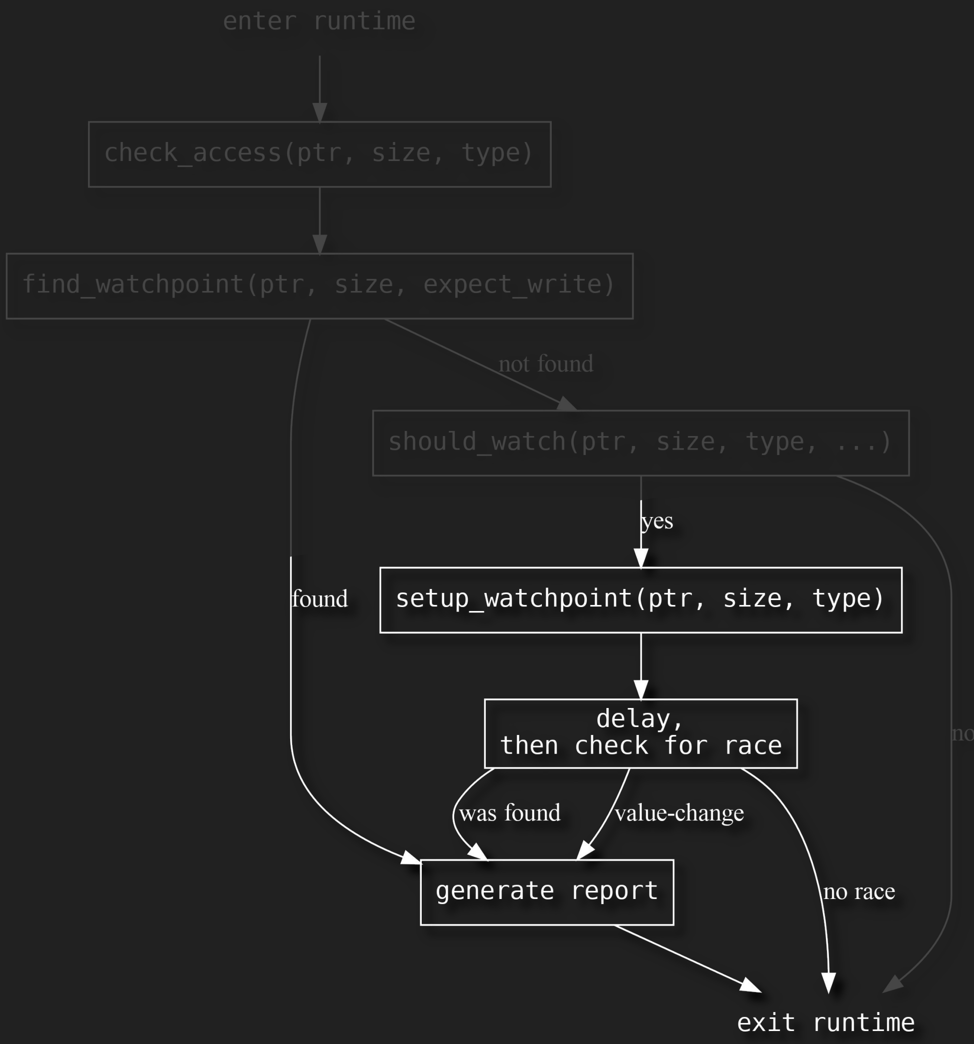
generate report

exit runtime

Implementing the memory model	`should_watch()`
plain accesses	✓ (sampling)
marked / atomic	✗

KCSAN: Runtime

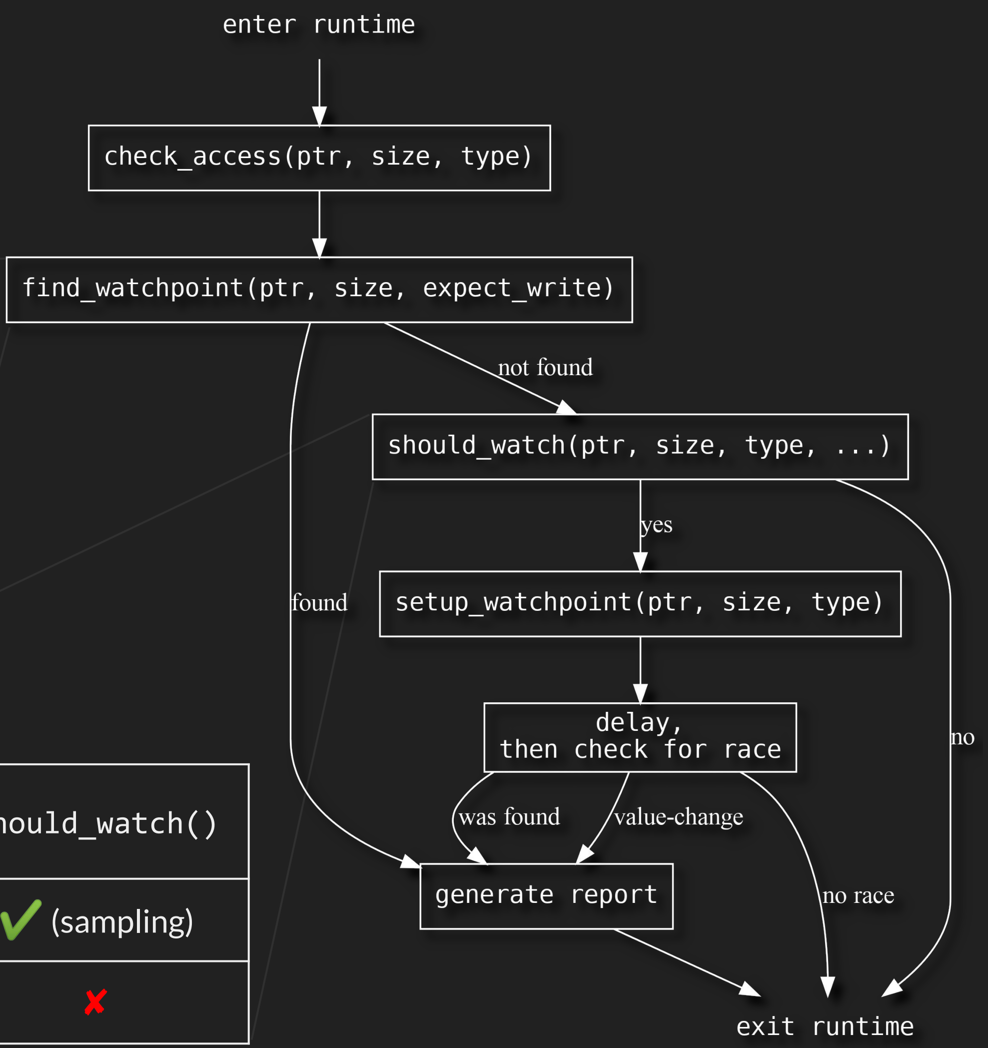
(slow path)



KCSAN: Runtime

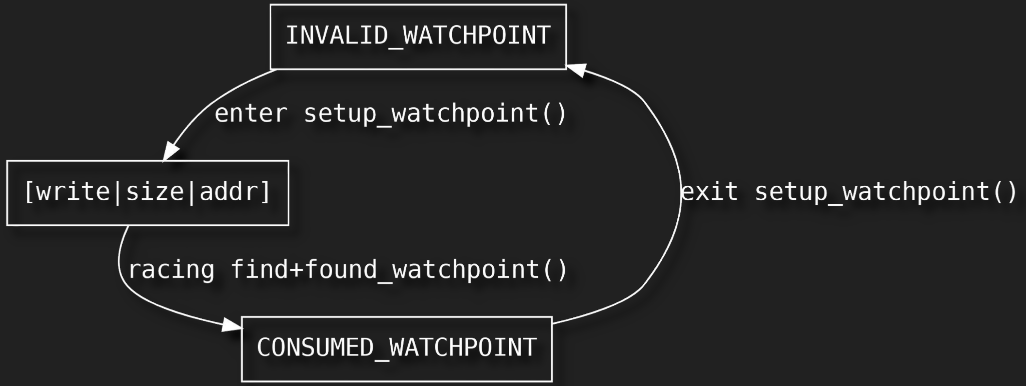
`find_watchpoint()`	read	write
read	✗	✓
write	✓	✓

Implementing the memory model	`should_watch()`
plain accesses	✓ (sampling)
marked / atomic	✗



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.



`INVALID_WATCHPOINT`

`enter setup_watchpoint()`

`[write|size|addr]`

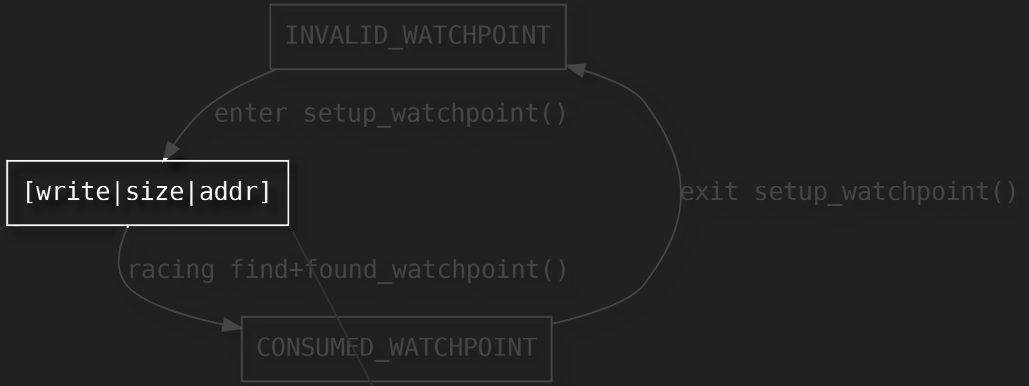
`exit setup_watchpoint()`

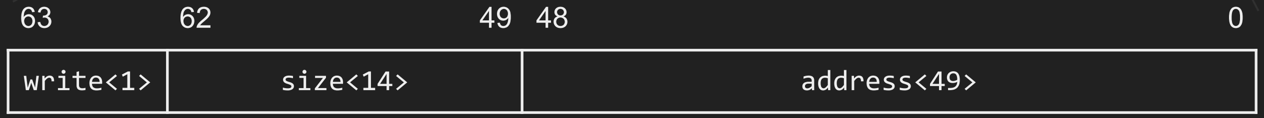
`racing find+found_watchpoint()`

`CONSUMED_WATCHPOINT`

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

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

```
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);
```

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);
```

Thread 2

```
/* Bug! */
WRITE_ONCE(shared_foo, 42);
```

Concurrency bugs that are not data races

Thread 0

```
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);
```

Thread 1

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

Thread 2

```
~~/* 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>
<tr><th></th><th colspan="2">concurrent writes</th><th>concurrent reads</th></tr>
<tr><td>ASSERT_EXCLUSIVE_WRITER(<i>var</i>)
ASSERT_EXCLUSIVE_WRITER_SCOPED(<i>var</i>)</td><td colspan="2">✗</td><td>✓</td></tr>
<tr><td>ASSERT_EXCLUSIVE_ACCESS(<i>var</i>)
ASSERT_EXCLUSIVE_ACCESS_SCOPED(<i>var</i>)</td><td colspan="2">✗</td><td>✗</td></tr>
<tr><td>ASSERT_EXCLUSIVE_BITS(<i>var</i>, <i>mask</i>)</td><td>~mask ✓</td><td>mask ✗</td><td>✓</td></tr>
</table>

Conclusion

Early community feedback and iterate

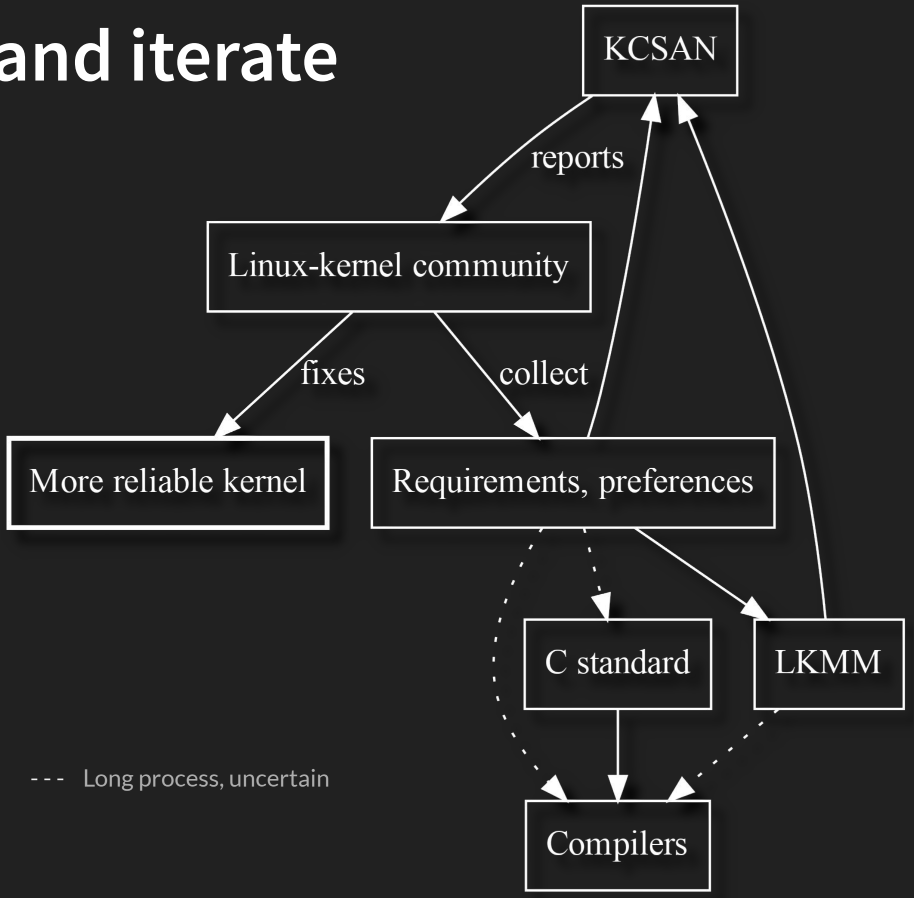
Examples:

- `CONFIG_KCSAN_REPORT_VALUE_CHANGE_ONLY`
- `data_race(...)` macro
- `CONFIG_KCSAN_ASSUME_PLAIN_WRITES_ATOMIC`

```
... = READ_ONCE(x) + 1;
```

```
x = 0xf0f0;
```

- `CONFIG_KCSAN_VERBOSE` (lockdep integration)



KCSAN

reports

Linux-kernel community

fixes

collect

More reliable kernel

Requirements, preferences

C standard

LKMM

Compilers

- - - Long process, uncertain

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`

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/

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