Applying Clang Thread Safety Analysis to the Linux Kernel

Lukas Bulwahn

Clang-Built Linux Workshop, February 2020

Outline

- Motivation
- Clang Thread Safety Analysis
- Existing Tools used in Linux
- Our Attempt and Results, incl. Random Impressions
- Suggestions for the Future

Motivation

- A next step after „building Linux with clang“:
  - Make use of all the small nice clang static analyses…
  - motivated by Nick‘s presentations and suggestions for future work
- Have an analysis during *kernel build time* that can guarantee that potentially concurrently accessible data cannot be accessed without locking (no race conditions)
- ~~Plan~~ Ilussion:
  - Run a new *static analysis* on the kernel…
  - Find thousands of bugs…
  - Fix them…
  - Become a hero

Real Plan:

- Find out what the tool can do
- Find a way to get some parts of the kernel covered
- Prove that it is maintainable
- Get piece by piece mainline
- Catch bugs when wrong changes happen…

Clang Thread Safety Analysis

- a C++ language extension in Clang, also works for C
- warns about potential race conditions in code
- developed by Google & CERT/SEI
- “like a type system for multi-threaded programs”
- based on checking developer annotations for each variable and function

Existing Tools used in Linux

- Many tools already out there for checking for concurrency issues:
  - sparse
  - smatch
  - coccinelle (mini_lock.cocci rule)
  - lockdep
  - KCSAN
  - coverity?
  - some tool from Jia-Ju Bai, Julia Lawall et al. (not open-source) [USENIX ATC ‘19]
  - …?

Himanshu‘s Report on tool capabilities:
https://github.com/himanshujha199640/linux-kernel-analysis/tree/report/gsoc19/reports

Being applied for years… So, we knew our real chances…

On my TODO list: have an overall kernel documentation explaining what all the different tools do for checking on concurrency issues.

include / linux / **compiler_attributes.h**

```
264  #if __has_attribute(capability)
265  # define __capability(x)                  __attribute__((capability(x)))
266  # define __acquires_mutex(x)              __attribute__((acquire_capability(x)))
267  # define __releases_mutex(x)              __attribute__((release_capability(x)))
268  # define __try_acquires_mutex(r, x)       __attribute__((try_acquire_capability(r, x)))
269  # define __requires_mutex(x)              __attribute__((requires_capability(x)))
```

include / linux / **mutex.h**

```
178  extern void mutex_lock(struct mutex *lock) __acquires_mutex(lock);
179  extern int __must_check mutex_lock_interruptible(struct mutex *lock) __try_acquires_mutex(0, lock);
196  extern int mutex_trylock(struct mutex *lock) __try_acquires_mutex(1, lock);
197  extern void mutex_unlock(struct mutex *lock) __releases_mutex(lock);
```

```
$ make -j1 HOSTCC=clang-8 CC=clang-8
CFLAGS_KERNEL="-Wthread-safety"' 2>&1 > /dev/null
```

Now just annotate *all functions*…

```
drivers/net/ethernet/realtek/r8169.c:740:1: warning: mutex 'tp->wk.mutex' is still held at the end of function [-Wthread-safety-analysis]
}
^
drivers/net/ethernet/realtek/r8169.c:739:2: note: mutex acquired here
        mutex_lock(&tp->wk.mutex);
        ^
drivers/net/ethernet/realtek/r8169.c:744:2: warning: releasing mutex 'tp->wk.mutex' that was not held [-Wthread-safety-analysis]
        mutex_unlock(&tp->wk.mutex);
        ^
```

4 drivers/net/ethernet/realtek/r8169.c

```
@@ -734,12 +734,12 @@ static inline struct device *tp_to_dev(struct rtl8169_private *tp)
734 734          return &tp->pci_dev->dev;
735 735    }
736 736
737     - static void rtl_lock_work(struct rtl8169_private *tp)
    737 + static void rtl_lock_work(struct rtl8169_private *tp) __acquires_mutex(tp->wk.mutex)
738 738    {
739 739          mutex_lock(&tp->wk.mutex);
740 740    }
741 741
742     - static void rtl_unlock_work(struct rtl8169_private *tp)
    742 + static void rtl_unlock_work(struct rtl8169_private *tp) __releases_mutex(tp->wk.mutex)
743 743    {
744 744          mutex_unlock(&tp->wk.mutex);
745 745    }
```

https://github.com/ClangBuiltLinux/thread-safety-analysis/commit/ccf6a9d69dc3de1e351f3025c8f59efb9d8a66f6

4 drivers/net/ethernet/realtek/r8169.c

```
@@ -734,12 +734,12 @@ static inline struct device *tp_to_dev(struct rtl8169_private *tp)
734 734         return &tp->pci_dev->dev;
735 735   }
736 736
737     - static void rtl_lock_work(struct rtl8169_private *tp)
    737 + static void rtl_lock_work(struct rtl8169_private *tp) __acquires_mutex(tp->wk.mutex)
738 738   {
739 739         mutex_lock(&tp->wk.mutex);
740 740   }
741 741
742     - static void rtl_unlock_work(struct rtl8169_private *tp)
    742 + static void rtl_unlock_work(struct rtl8169_private *tp) __releases_mutex(tp->wk.mutex)
743 743   {
744 744         mutex_unlock(&tp->wk.mutex);
745 745   }
```

```
static int __d_unalias(struct inode *inode,
                struct dentry *dentry, struct dentry *alias)
{
        struct mutex *m1 = NULL;
        struct rw_semaphore *m2 = NULL;
        int ret = -ESTALE;

        /* If alias and dentry share a parent, then no extra locks required */
        if (alias->d_parent == dentry->d_parent)
                goto out_unalias;

        /* See lock_rename() */
        if (!mutex_trylock(&dentry->d_sb->s_vfs_rename_mutex))
                goto out_err;
        m1 = &dentry->d_sb->s_vfs_rename_mutex;
        if (!inode_trylock_shared(alias->d_parent->d_inode))
                goto out_err;
        m2 = &alias->d_parent->d_inode->i_rwsem;
out_unalias:
        __d_move(alias, dentry, false);
        ret = 0;
out_err:
        if (m2)
                up_read(m2);
        if (m1)
                mutex_unlock(m1);
        return ret;
}
```

fs / dcache.c

```
static int __d_unalias(struct inode *inode,
                struct dentry *dentry, struct dentry *alias)
{
        /* If alias and dentry share a parent, then no extra locks
        if (alias->d_parent == dentry->d_parent) {
                __d_move(alias, dentry, false);
                return 0;
        }
        /* See lock_rename() */
        if (!mutex_trylock(&dentry->d_sb->s_vfs_rename_mutex))
                return -ESTALE;
        if (!inode_trylock_shared(alias->d_parent->d_inode)) {
                mutex_unlock(&dentry->d_sb->s_vfs_rename_mutex);
                return -ESTALE;
        }
        __d_move(alias, dentry, false);
        up_read(&alias->d_parent->d_inode->i_rwsem);
        mutex_unlock(&dentry->d_sb->s_vfs_rename_mutex);
        return 0;
}
```

Our attempts and results

Annotations on mutex primitives (around August 2019)

- 208 effective annotations
- 98 silencing annotations
- ~150 remaining warnings on defconfig

Annotations on spinlock primitives (by Himanshu Jha)

- 77 effective annotations
- 108 silencing annotations
- 281 remaining warnings on defconfig

Investigation on spinlocks, done by Himanshu Jha (GSoC student 2019):

https://github.com/ClangBuiltLinux/thread-safety-analysis/tree/clang-thread-safety-analysis-spinlock

Investigation on mutexes:

https://github.com/ClangBuiltLinux/thread-safety-analysis/tree/clang-thread-safety-analysis-v3

Recording False Positives

https://github.com/clangbuiltlinux/thread-safety-analysis/issues

conditionally held locks	Known limitation of thread safety analysis	56 open issues
aliasing	analysis does not handle aliases	21 open issues
annotate return value	We would need to annotate and refer to the function's return value	4 open issues
special locking/unlocking pattern	Code uses a special pattern to lock or unlock	4 open issues

conditionally held locks Known limitation of thread safety analysis 56 open issues

```
- static inline void trace_access_lock(int cpu)
+ static inline void trace_access_lock(int cpu) __acquires_mutex(cpu_access_lock) __no_thread_safety_analysis
  {
        if (cpu == RING_BUFFER_ALL_CPUS) {
                /* gain it for accessing the whole ring buffer. */
                down_write(&all_cpu_access_lock);
        } else {
                /* gain it for accessing a cpu ring buffer. */

                /* Firstly block other trace_access_lock(RING_BUFFER_ALL_CPUS). */
                down_read(&all_cpu_access_lock);

                /* Secondly block other access to this @cpu ring buffer. */
                mutex_lock(&per_cpu(cpu_access_lock, cpu));
        }
  }
```

drivers/base/core.c

conditionally held locks

Known limitation of thread safety analysis

56 open issues

```
3080                    if (parent)
3081                            device_lock(parent);
3082                    device_lock(dev);
```

```
3103                    device_unlock(dev);
3104                    if (parent)
3105                            device_unlock(parent);
```

Suggestions for the Future

- Try other static analysis tools based on clang…
- Improve clang thread safety analysis
- Slowly get more annotations in the kernel (consolidate annotations and piggy-back on the existing sparse&lockdep annotations)
  - Understand why did interest in sparse annotations fade away?

- We actually never annotated variables with __guarded_by(<lock>)

Simple Improvements #1

- Allow configuring if unbalances due to certain callers shall be warned about, e.g., have annotations to warn at a certain warning level
  - Annotate lock & unlock to let the analysis know they acquire & release, but do not warn about all unbalanced functions due to lock & unlock
  - Only warn if the further functions (users) are actually annotated wrong

=> No need to silence all false positives with useless annotations

Simple Improvements #2

- Increase details of reporting on the analysis results (function coverage of analysis)
  - How many functions are analysed with a given set of annotations?
  - Which specific annotation contributes to the analysis of other specific functions?
  - How many functions are ignored due to "no thread safety analysis" annotations in the whole build?
  - Which annotations have no further impact beyond the local scope of the annotated function?

Get more annotations in the kernel

Patches with __no_thread_safety_analysis are not going to be accepted…

… but if there are USEFUL annotations for one tool (e.g. a coccinelle rule) to classify and check certain classes of functions, e.g. __conditionally_acquire(…) …

… Just use those for the other tools to NOT report warnings on such annotations (false positive annotations in disguise…)

Conclusion

Clang Thread Safety Analysis:

- Nice small experiment... Easy setup… runs quickly…
- It is suitable for students/mentees to work on…
- Not the most promising results though…

Will I ever get back to annotate everything to reach zero warnings on defconfig?

Suggestions for alternative tools to look into?

Thank you!

Many thanks to

- *Nick Desaulniers* for motivating this work and providing a home at the Clang-Built Linux github organisation
- *Himanshu Jha* for his work in this investigation
- *Google Summer of Code program* to fund Himanshu's work
- *the Linux Foundation* for serving as umbrella organisation in the GSoC program
- *Arnd Bergmann, Neil Brown and Nicholas McGuire* for review and discussion of two RFC patch proposals (one patch where the tool completely confused me and I send a patch breaking things rather than fixing them)
- *Google‘s Open-Source Program Office* for the invitation to this workshop