Paul E. McKenney, Facebook

Linux Plumbers Conference: Toolchains & Kernel MC, September 24, 2021

Report From The Standards Committees

What Has Been Happening?

- Concurrency TS 2 (hazard pointers, RCU)
- Lifetime-end pointer zap
- Undefined behavior
- Relaxed guide to relaxed
- volatile_load and volatile_store
- Address/data dependency ordering
  - Control dependencies covered in next session (“The never-ending saga of control dependencies”)

Concurrency Technical Specification 2

Concurrency Technical Specification 2

- In June 2021, C++ plenary session requested a Concurrency TS 2:
  - Hazard pointers
  - RCU: Adjusted to allow bare-bones implementations
  - Asymmetric fences? (sys_membarrier())
- Maybe into C++26 or C++29

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p1121r3.pdf
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p1122r4.pdf

Concurrency TS 2: RCU Adjustments

- Naming (e.g., “rcu_synchronize()”)
- No rcu_head: Instead inherit from rcu_obj_base template class
- “Non-intrusive” retire() (AKA call_rcu() in kernel)
  - Zero storage overhead, similar to single-argument kvfree_rcu() in kernel
- Callback invocation from retire()
  - Allows use in constrained environments, as in without softirq or any RCU grace-period kthread
- RAII readers: Automatic rcu_read_unlock() at end of scope
  - There are mechanisms to allow explicit unlock

Shameless plug: https://cppcon.org/ presentation week of October 24th with Maged Michael and Michael Wong.

Lifetime-End Pointer Zap

Lifetime-End Pointer Zap

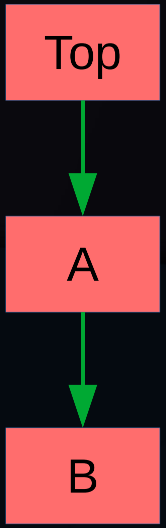
- Important concurrent algorithms intentionally access pointers to lifetime-ended objects
  - LIFO Push (similar to Treiber’s stack ca. 1973)
    - Single-element push and pop-all operations
  - Hazard pointers
  - Variants of sharded-locking methodology
  - Any number of debugging schemes

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p1726r5.pdf
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2414r1.pdf

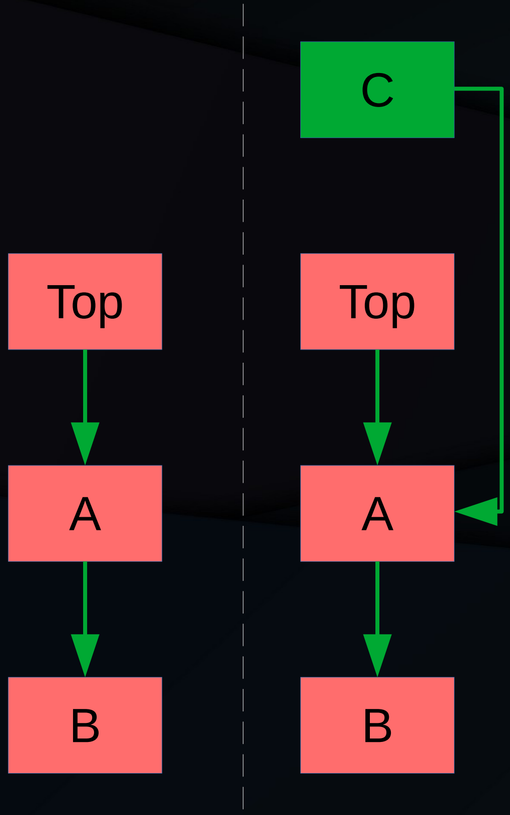
LIFO Push Algorithm Outline

- Push a single element
  - Allocate and initialize data fields
  - Repeat until `cmpxchg()` succeeds:
    - Initialize `->next` pointer to `top` pointer
    - Use `cmpxchg()` to point `top` to new element
- Pop entire stack
  - Use `xchg()` to pop entire list, setting `top` to `NULL`

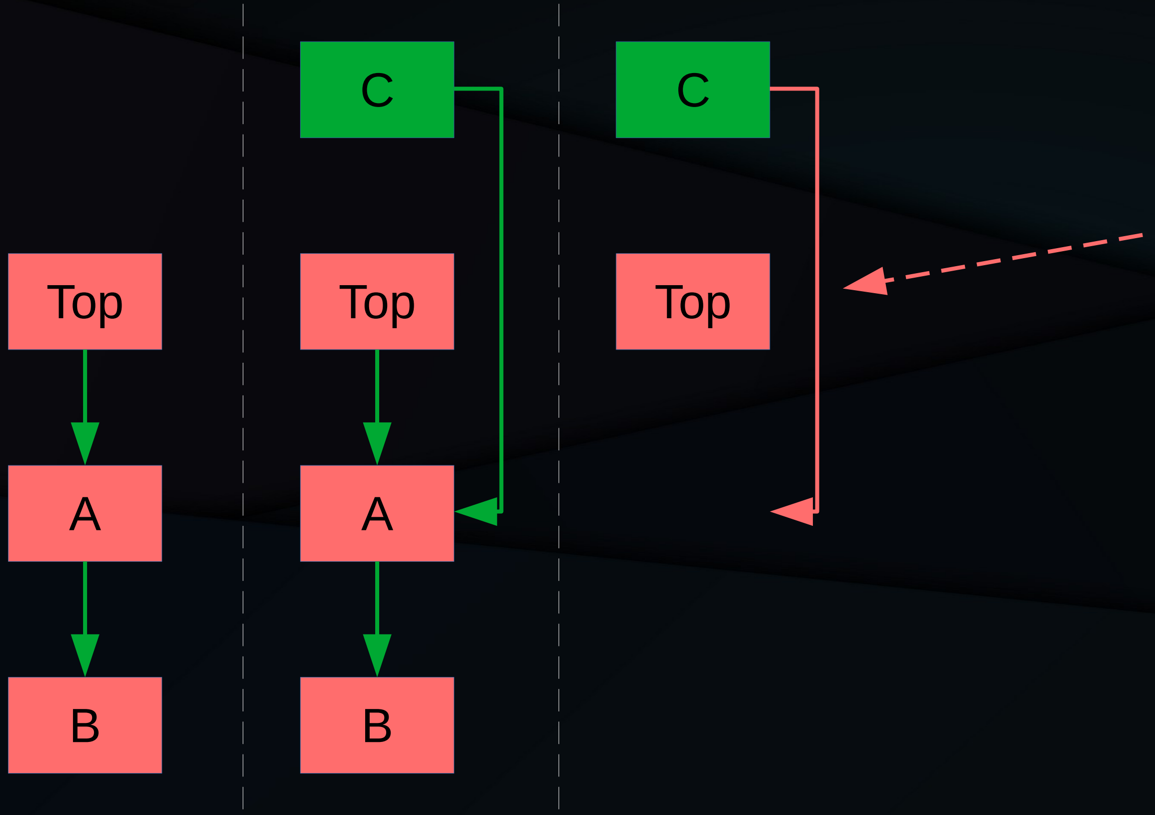
Initial State (Red == Concurrency)



Push of C Begin: Allocate & Initialize

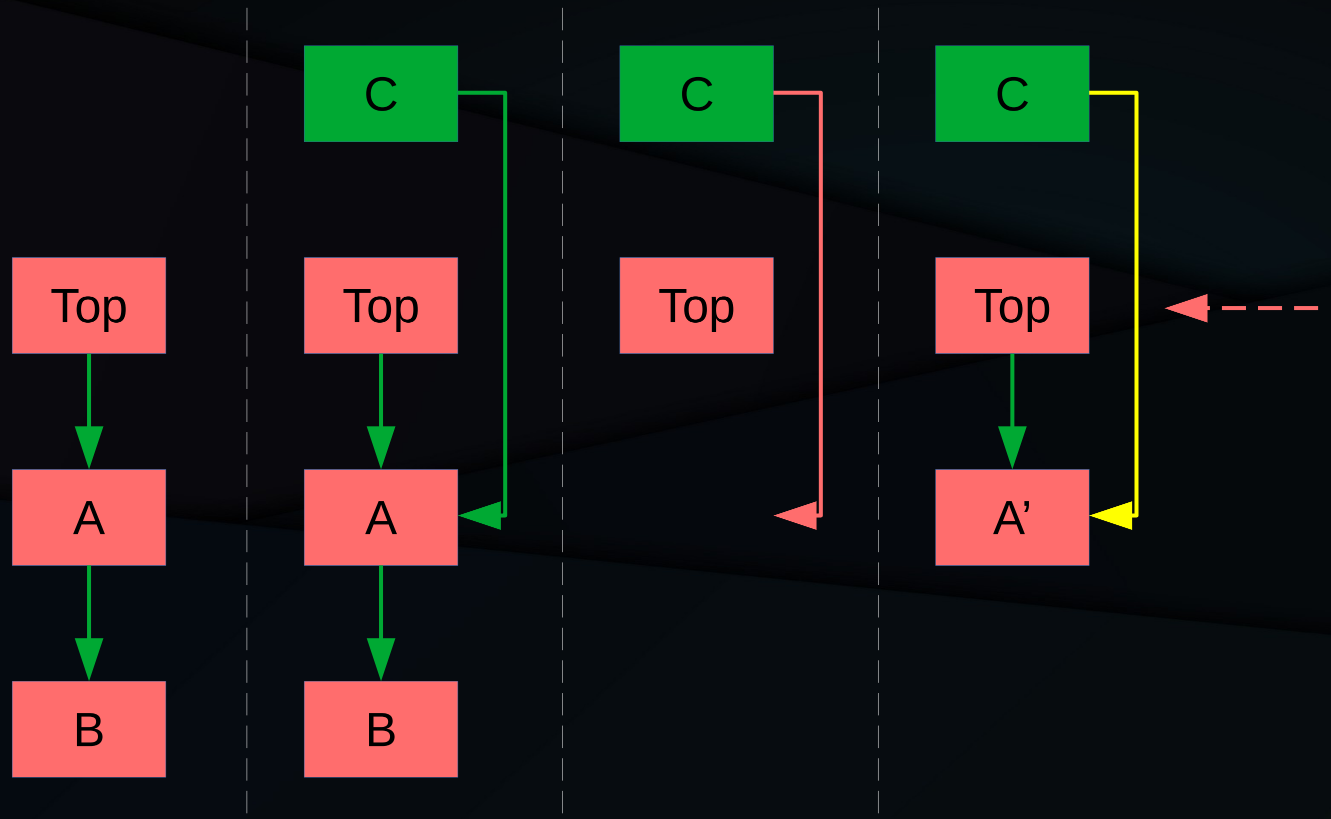


Intervening Pop-All Operation!!!



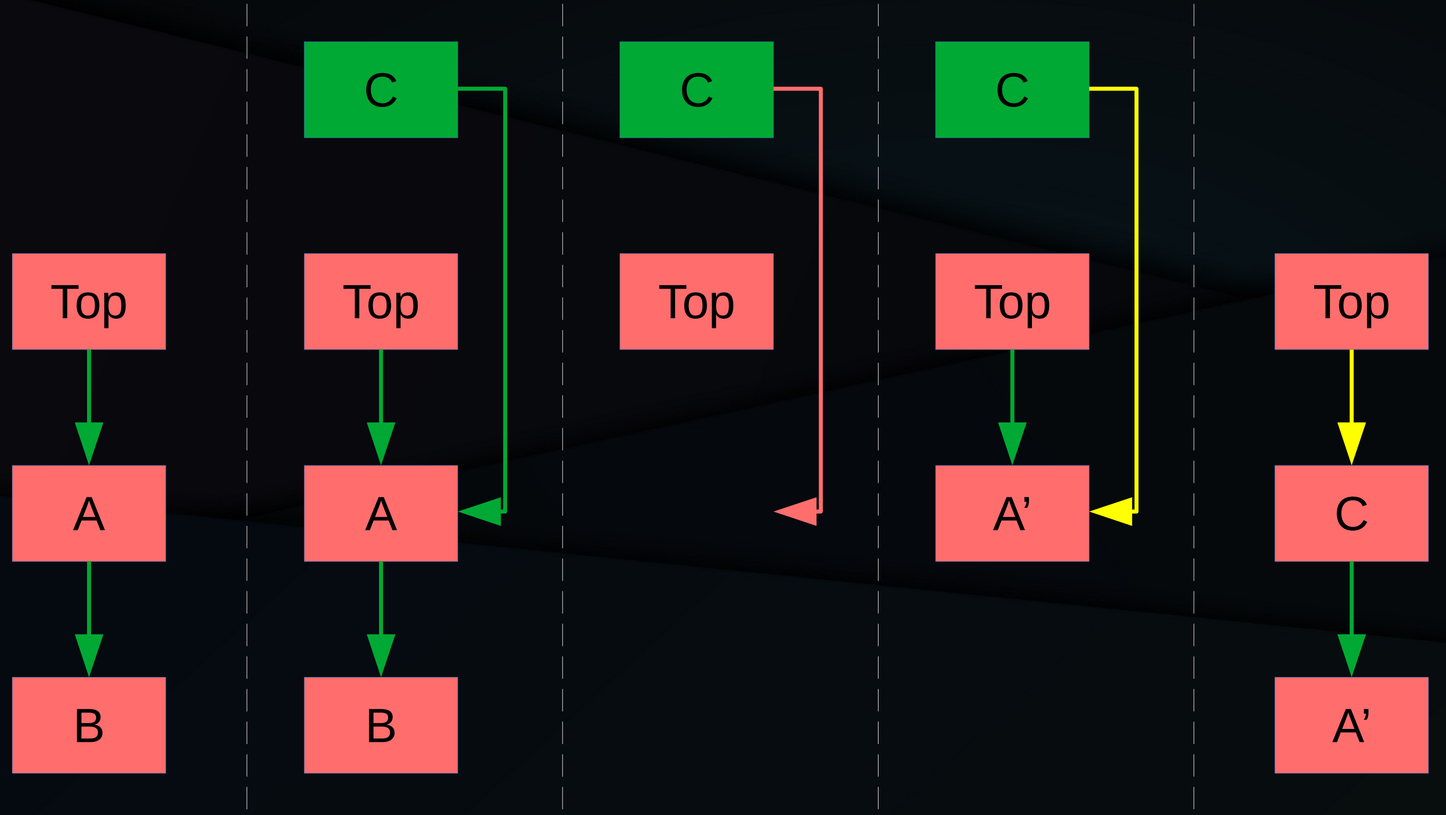
“Indeterminate” pointer (C) or “Invalid” pointer (C++) to lifetime-ended object

Push of A’ (Reuses Memory of A)

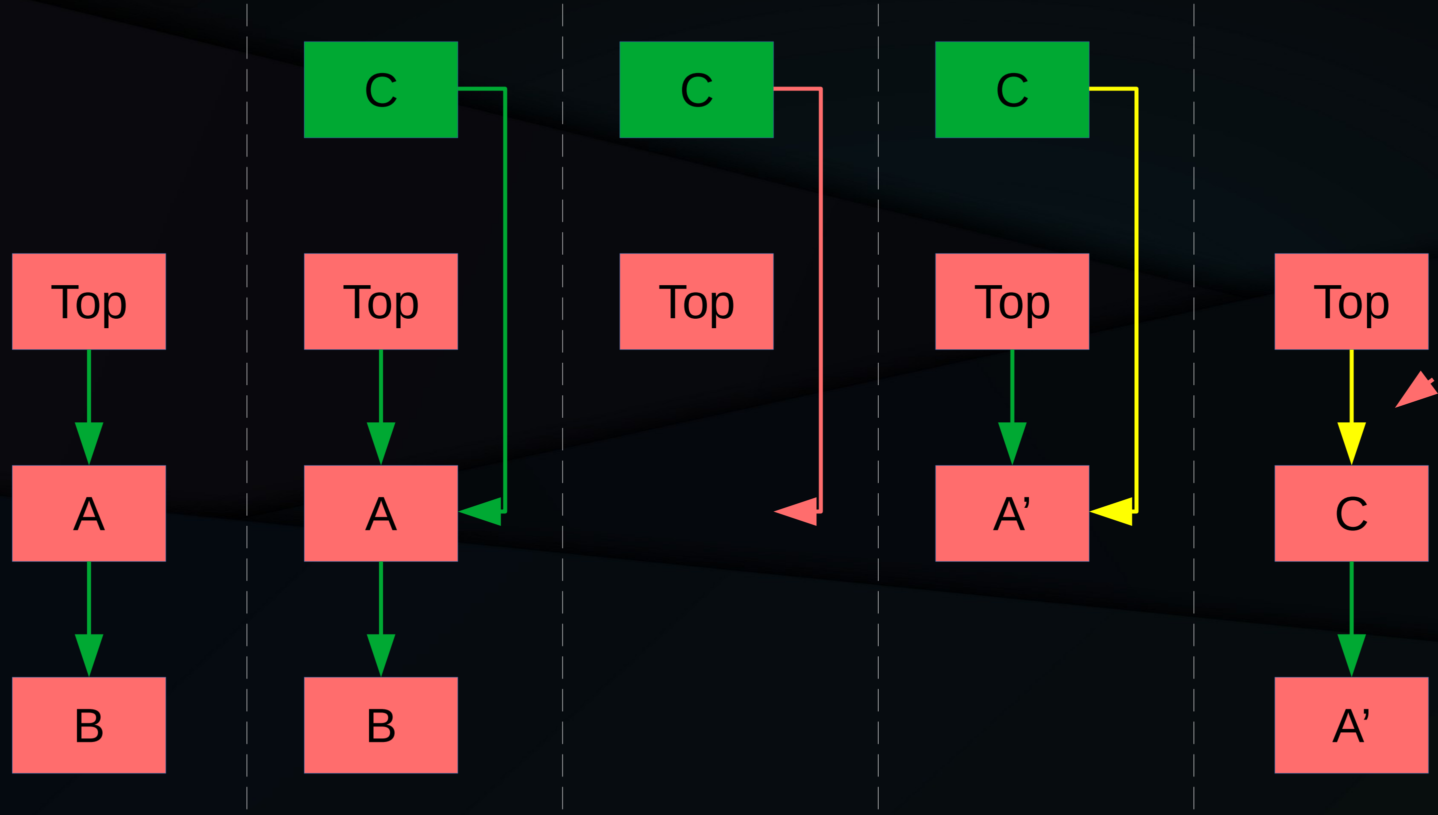


“Zombie pointer” to newly allocated type-compatible object A’ that happens to have the same address that object A used to have.

Push of C Finally Completes



Compilers Hate Zombie Pointers!!!



The list’s bits are just fine, but the compiler hates this zombie pointer!!!

Undefined Behavior

Undefined Behavior (UB)

- UB can back-propagate
- UB anywhere? Undefined everywhere!

```
int a[5];
int i = 3;


a[i] = 5;
```

Array-Out-Of-Bounds UB

- UB can back-propagate
- UB anywhere? Undefined everywhere!

```
int a[5];
int i = 3;


a[i] = 5;
```

UB here might set “i” to 5...

Back-Propagated UB

- UB can back-propagate
- UB anywhere? Undefined everywhere!

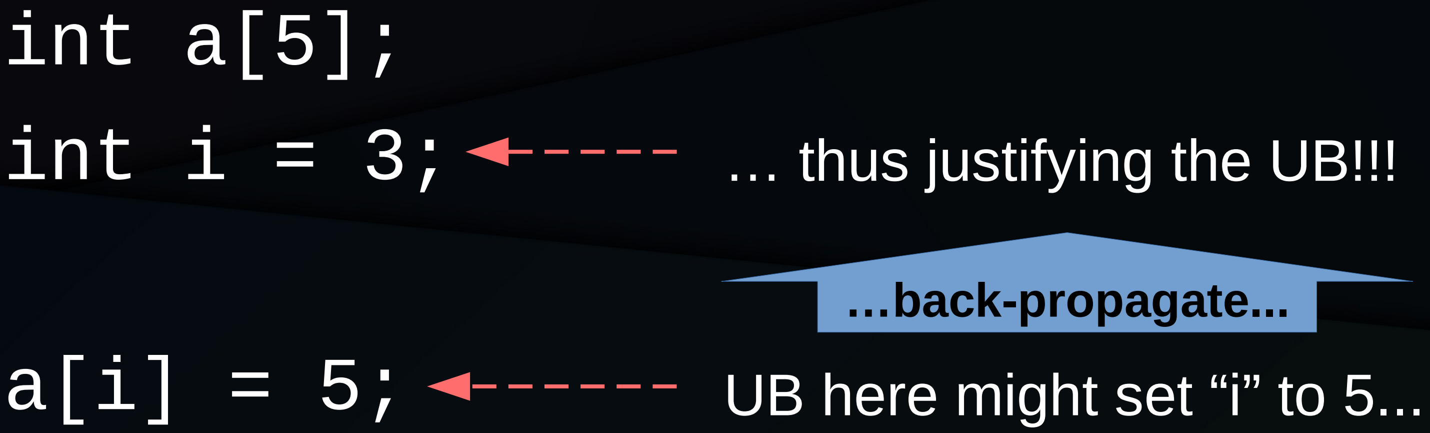
```
int a[5];
int i = 3;


a[i] = 5;
```

Back-propagate...

UB here might set “i” to 5...

Self-Justifying UB (Anonymous, 2007)

- UB can back-propagate
- UB anywhere? Undefined everywhere!



```
int a[5];
int i = 3;


a[i] = 5;
```

… thus justifying the UB!!!

…back-propagate...

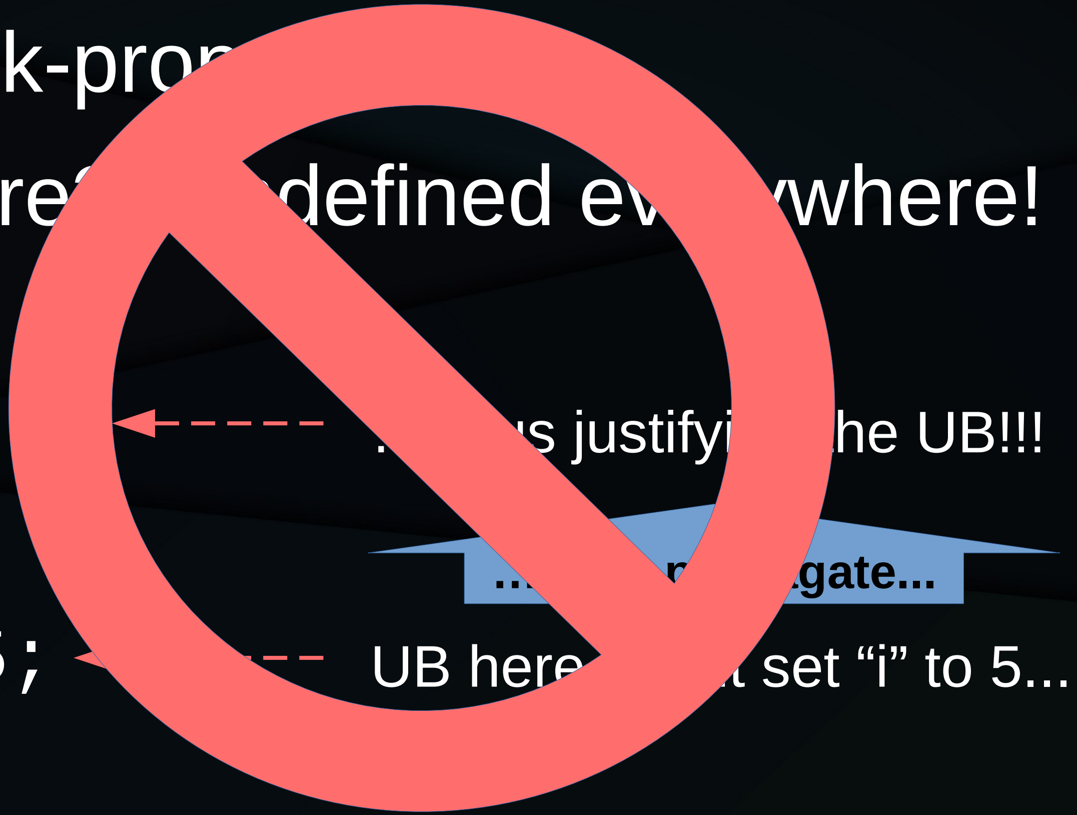
UB here might set "i" to 5...

Anonymous due to Chatham House rules. Eminent academic.

Self-Justifying UB (Anonymous, 2007)

- UB can back-pro
- UB anywhere defined ev ywhere!

```
int a[5]
int i =


a[i] = 5;
```

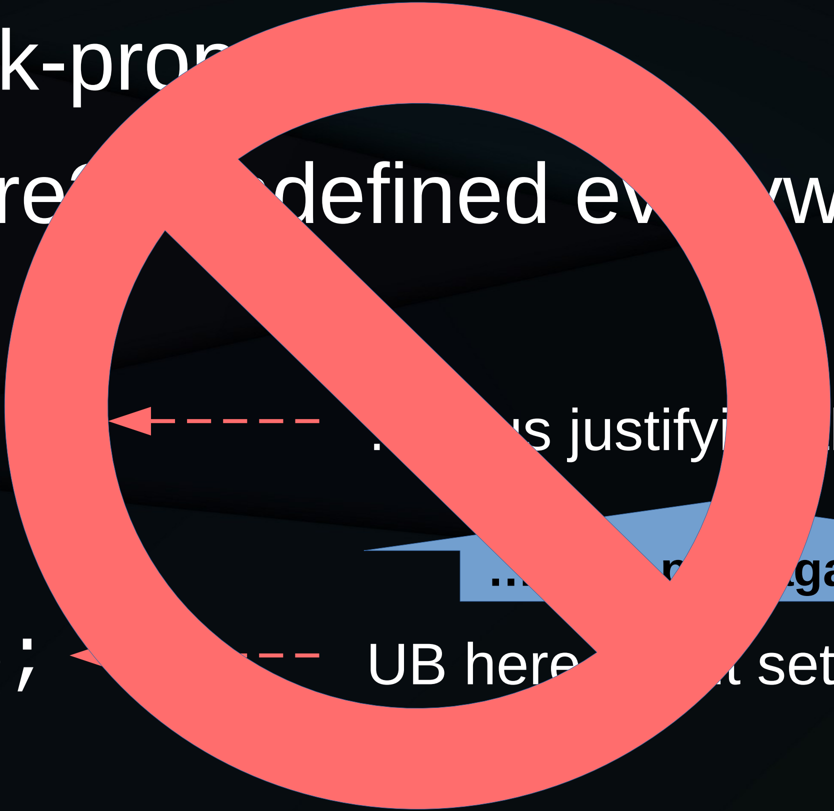


“Cannot happen”, but no formal limitation justifying this.

Self-Justifying UB (Anonymous, 2007)

- UB can back-prop
- UB anywhere defined ev ywhere!

```
int a[5]

int i =


a[i] = 5;
```



. s justify he UB!!!

... gate...

UB here t set “i” to 5...

“Cannot happen”... Unless `memory_ordered_relaxed` concurrency is in play.

Relaxed Guide to Relaxed

Relaxed Guide to Relaxed

- C/C++ memory model allows OOTA values
  - T1: `x.store(y.load, mo_relaxed), mo_relaxed);`
  - T2: `y.store(x.load, mo_relaxed), mo_relaxed);`
  - Even if x & y initially `0`, could have `x==y==42`
- Why? C & C++ don’t respect dependencies!

Relaxed Guide to Relaxed

- C/C++ memory model allows OOTA values
  - T1: `x.store(y.load, mo_relaxed), mo_relaxed);`
  - T2: `y.store(x.load, mo_relaxed), mo_relaxed);`
  - Even if `x` & `y` initially `0`, could have `x==y==42`
- Why? C & C++ don’t respect dependencies!
  - Random `mo_relaxed` programs problematic...

This can get quite ugly: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1916r0.pdf`

Relaxed Guide to Relaxed

- Random `mo_relaxed` programs problematic?
  - Don’t ever use `memory_order_relaxed`!!!

Relaxed Guide to Relaxed

- Random `mo_relaxed` programs problematic?
  - Don’t ever use `memory_order_relaxed`!!!
    - Pity about the poor performance...

Relaxed Guide to Relaxed

- Random `mo_relaxed` programs problematic?
  - ~~Don't ever use `memory_order_relaxed`!!!~~
    - Pity about the poor performance...
  - Instead, don't randomly generate programs involving `memory_order_relaxed` accesses!!!

Relaxed Guide to Relaxed

- Random `mo_relaxed` programs problematic?
  - ~~Don't ever use `memory_order_relaxed`!!!~~
    - Pity about the poor performance...
  - Instead, don't randomly generate programs involving `memory_order_relaxed` accesses!!!
- Design programs using known-good patterns

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2055r0.pdf

volatile_load<T> & volatile_store<T>

volatile_load<T> & volatile_store<T>

- Maybe C++’s answer to READ_ONCE() and WRITE_ONCE()

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1382r1.pdf

Address/Data Dependency Ordering

Address/Data Dependency Ordering

- Lots of electrons burned on this one…
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0371r1.html
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0098r1.pdf
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0462r1.pdf
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0190r4.pdf
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0750r1.html
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0735r1.html
- Next step: Implementation (GSoC prototype)

Summary

Summary

- C11/C++11 got the concurrency ball rolling
- But these cannot be the final word
- The historical separation of the C/C++ and concurrency communities has bitten us extremely hard!