Writing Performant Concurrent Data Structures

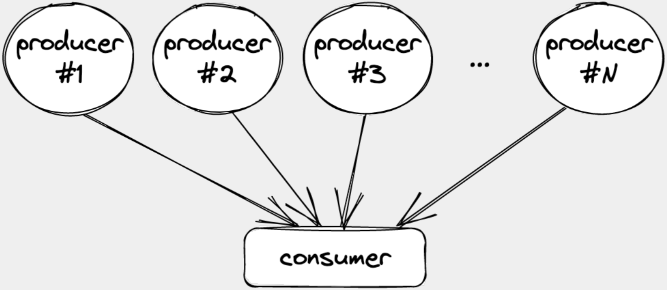
Adrian Alic

Software Engineer @ DFINITY
Website: `https://alic.dev`
Contact: `contact@alic.dev`

Rust Meetup Zürich
March 28, 2023

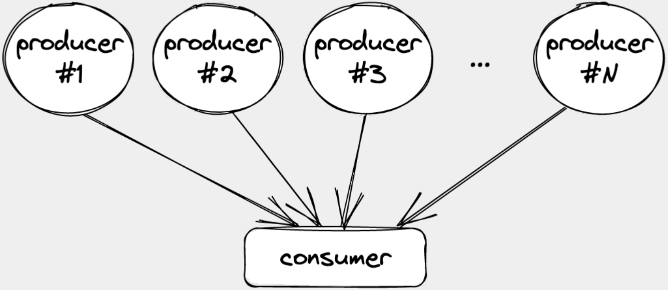
Overview

Case-study: Multi-producer, single-consumer queue.



Overview

Case-study: Multi-producer, single-consumer queue.



Goals:

- How to write such a queue
- How to make it fast
- How to reason about correctness

Motivation

A Multi-Core Logger

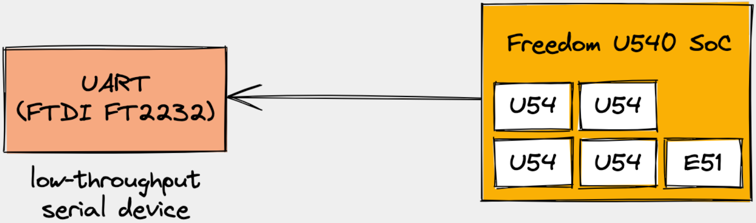


Figure: A sketch of a 5-core RISC-V SoC.

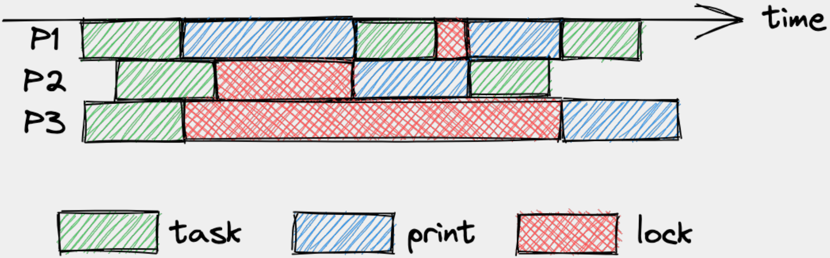
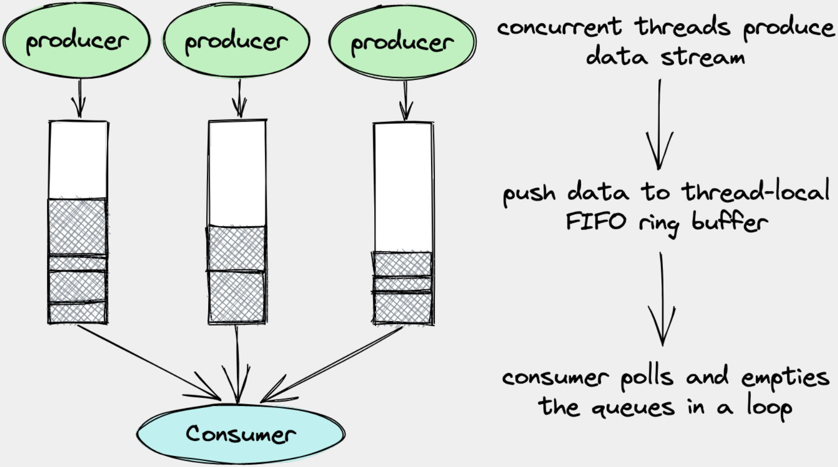


Figure: Locking causes unpredicable latency jitter.

The Idea

A Bunch of Ring Buffers



Naive Rust Definition

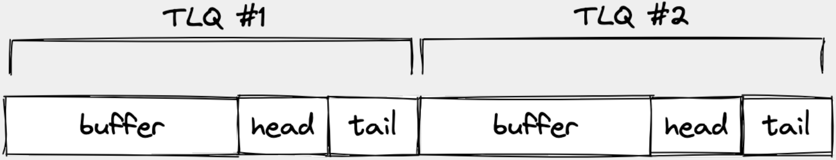
```rust
// if you like pointer indirection
struct TLQ {
        buffer: Vec<u8>,
        head: u16,
        tail: u16,
}

// if buffer size is known at compile-time
struct TLQ<const C: usize> {
        buffer: [u8; C],
        head: u16,
        tail: u16,
}
```

However: this definition has some problems...

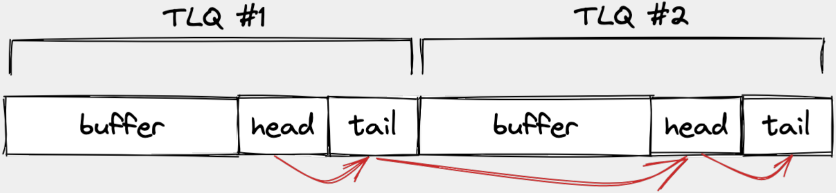
Lack of Cache Locality

If we store *multiple* TLQs in an array, iterating over heads and tails becomes costly.



Lack of Cache Locality

If we store *multiple* TLQs in an array, iterating over heads and tails becomes costly.



This problem of traversing fields is common in game development (ECS).

Improving Cache Locality

One solution: *Struct of Arrays*.

Improving Cache Locality

One solution: *Struct of Arrays*.

```
struct Offset {
        head: u16,
        tail: u16,
}
```

Improving Cache Locality

One solution: *Struct of Arrays*.

```
struct Offset {
        head: u16,
        tail: u16,
}

struct Buffer<const C: usize> {
        buffer: [u8; C],
}
```

Improving Cache Locality

One solution: *Struct of Arrays*.

```
struct Offset {
        head: u16,
        tail: u16,
}

struct Buffer<const C: usize> {
        buffer: [u8; C],
}

struct Queue<const T: usize, const C: usize> {
        offsets: [Offset; T],
        buffers: [Buffer<C>; T]
}
```

New Layout Visualized

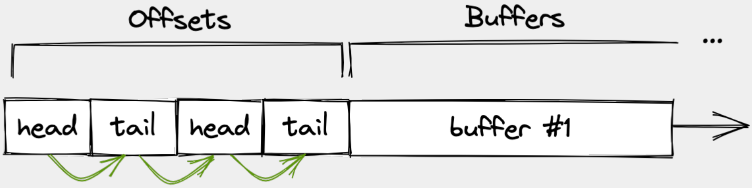


Figure: Our consumer can now iterate through all offsets without tons of cache misses.

Some languages like Zig have built-in support for the SoA pattern[1].

[1]https://kristoff.it/blog/zig-multi-sequence-for-loops/

The Memory Model

The Illusion of Safety on x86

Figure: Don't do this. The memory ordering I chose for my atomic ops only worked on x86, but blew up on a *weaker* memory model (aarch64).

Segfaults on aarch64

	Property	Alpha	Armv7-A/R	Armv8	Itanium	MIPS	POWER	SPARC TSO	x86	z Systems
Memory Ordering	Loads Reordered After Loads or Stores?	Y	Y	Y	Y	Y	Y			
	Stores Reordered After Stores?	Y	Y	Y	Y	Y	Y			
	Stores Reordered After Loads?	Y	Y	Y	Y	Y	Y	Y	Y	Y
	Atomic Instructions Reordered With Loads or Stores?	Y	Y	Y		Y	Y			
	Dependent Loads Reordered?	Y								
	Dependent Stores Reordered?									
	Non-Sequentially Consistent?	Y	Y	Y	Y	Y	Y	Y	Y	Y
	Non-Multicopy Atomic?	Y	Y	Y	Y	Y	Y	Y	Y	
	Non-Other-Multicopy Atomic?	Y	Y		Y	Y	Y			
	Non-Cache Coherent?				Y					

Figure: McKenney [1, p. 352] lists differences between hardware platforms in detail.

C11 Memory Model

Rust follows the C11 memory ordering spec[2]. It includes:

[2] https://en.cppreference.com/w/cpp/atomic/memory_order

C11 Memory Model

Rust follows the C11 memory ordering spec[2]. It includes:

Specification of *modification order*:

- RR/RW/WR/WW Coherency

Flavors of "before":

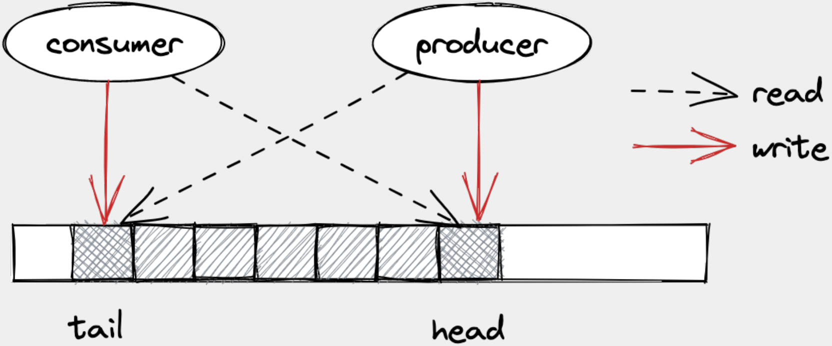
- Sequenced-before
- Dependency-ordered before
- Inter-thread happens-before
- Happens-before

Also relevant: *evaluation order*[3]

---

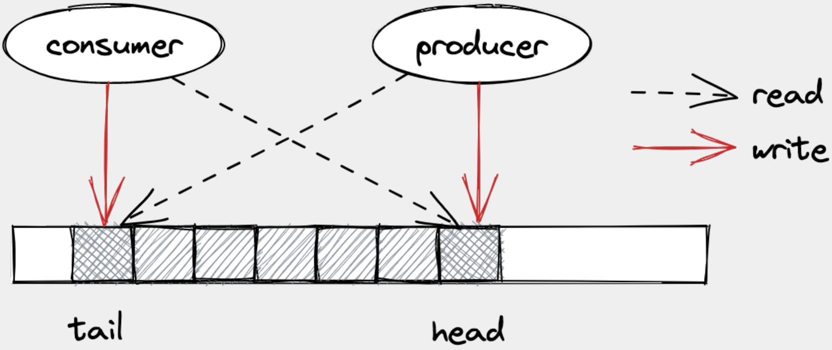
[2] https://en.cppreference.com/w/cpp/atomic/memory_order
[3] https://en.cppreference.com/w/cpp/language/eval_order

Concurrency Behavior of Our Queue



[4] https://doc.rust-lang.org/std/sync/atomic/struct.AtomicU64.html#method.compare_exchange

Concurrency Behavior of Our Queue



Our queue is essentially an SPSC without competing stores - thus we have no need for atomic RMW primitives[4].

[4] https://doc.rust-lang.org/std/sync/atomic/struct.AtomicU64.html#method.compare_exchange

The Two Basic Queue Operations

Our SPSC requires two release-acquire pairs. We can look at the first one below.

```
// producer thread
fn push(data) {
  h = head.load(_)
  new_h = h + data.len()

  // write data
  buffer[h..new_h] = data;

  // update index
  h.store(new_h, _)
}
```

```
// consumer thread
fn pop() [u8] {
  // read index
  h = tail.load(_)
  t = tail.load(_)

  // read data
  buffer[t..h]
}
```

The Two Basic Queue Operations

Our SPSC requires two release-acquire pairs. We can look at the first one below.

```
// producer thread
fn push(data) {
  h = head.load(_)
  new_h = h + data.len()

  // write data
  buffer[h..new_h] = data;

  // update index
  h.store(new_h, release)
}
```

```
// consumer thread
fn pop() [u8] {
  // read index
  h = tail.load(acquire)
  t = tail.load(_)

  // read data
  buffer[t..h]
}
```

Implementation in Rust

Avoiding False Sharing

Since offsets are accessed concurrently, we need to be aware of cache coherence effects.

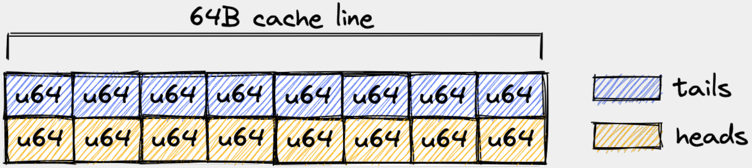


Figure: The most common solution is to pad all shared fields to a cache line.

Cache-Alignment for Each Offset

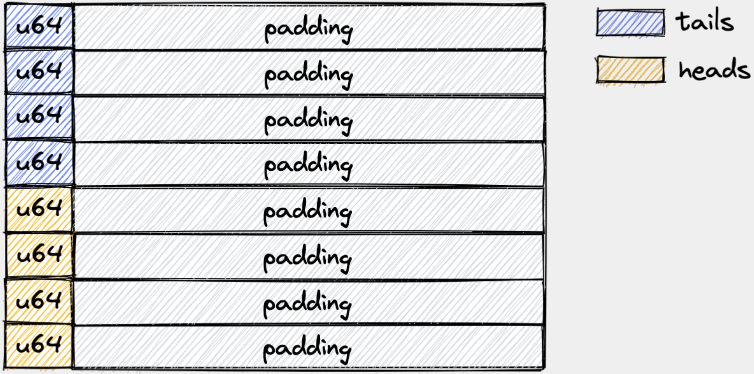


Figure: Fully padded version. No false sharing will occur.

A Possible Middle Ground

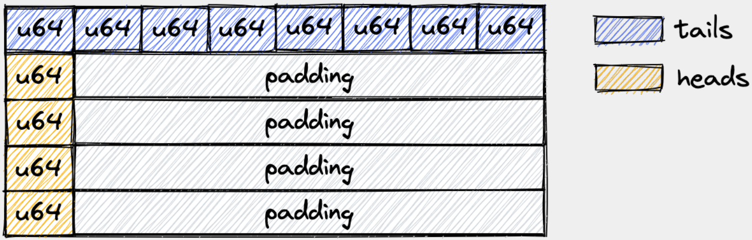


Figure: This hybrid version allows for atomic batch updates.

Implementation in Rust

```rust
#[repr(align(64))]
struct Tail(u16);

#[repr(align(64))]
struct Head(u16);

struct Offsets<const T: usize> {
    tails: [Tail; T],
    heads: [Head; T],
}

// Or alternatively, use the crossbeam_util crate
struct Offsets<const T: usize> {
    tails: [CachePadded<Tail>; T],
    heads: [CachePadded<Head>; T],
}
```

False Sharing Can Have a Large Impact

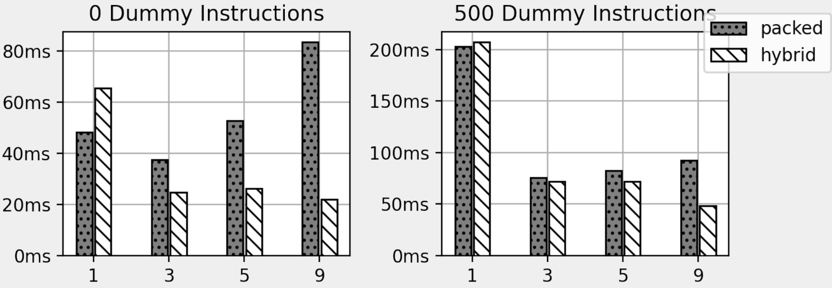


Figure: From a benchmark on false sharing [5]

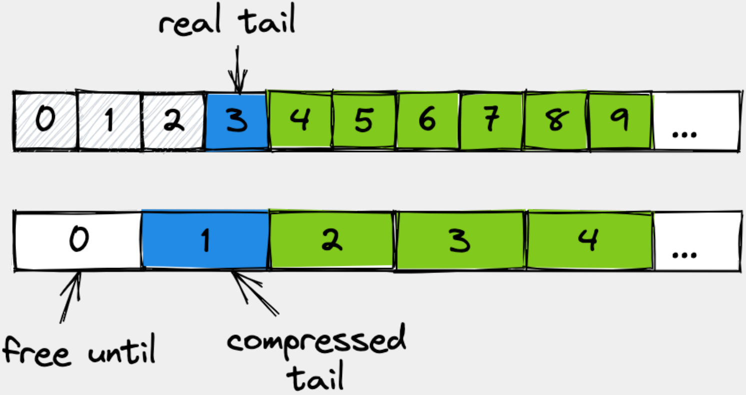
.

[5] https://alic.dev/blog/false-sharing

Consumer-Side Pointer Compression

Figure: We can decrease the addressing granularity, reducing memory footprint.

Pointer Compression Visualized

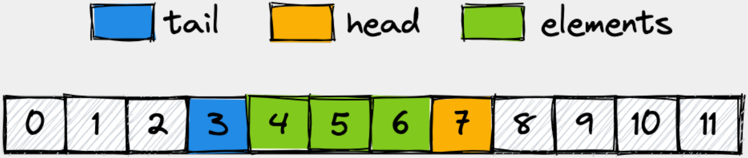


Implementation in Rust

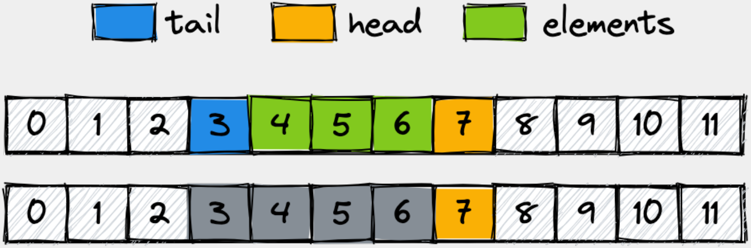
```rust
struct Consumer<const C: usize> {
    shared_tail: *const AtomicU16,
    local_tail: usize,
}
fn update_tail(&mut self, val) {
    self.local_tail = val;
    self.shared_tail.store(
        compress(self.local_tail, C),  // <---
        Ordering::Release
    );
}
fn compress(tail: usize, C: usize) -> u16 {
    let shift = if C <= 16 { 0 } else { C - 16 };
    (tail >> shift)
}
```

Local Caching of Offsets

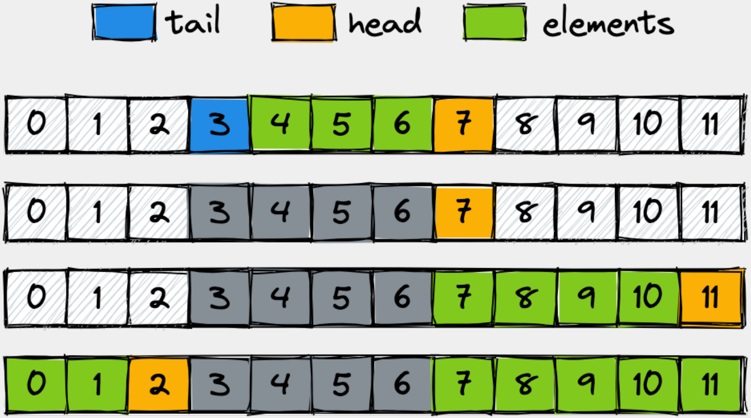
Local Caching of Offsets



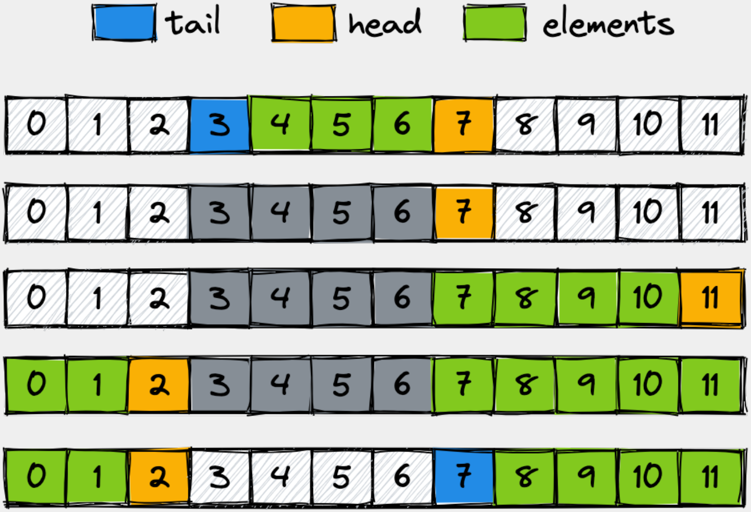
Local Caching of Offsets



Local Caching of Offsets



Local Caching of Offsets



Crafting Safe Abstractions

Limits of the Borrow Checker

The borrow checker and lifetime system is not designed to reason about correctness of arbitrary concurrent data structures.

Example: *Atomics*

```
impl AtomicUsize {
    pub fn store(&self, val: bool, order: Ordering) {
        // SAFETY: any data races are prevented by atomic
        // intrinsics and the raw pointer passed in is
        // valid because we got it from a reference.
        unsafe {
            atomic_store(self.v.get(), val as u8, order);
        }
    }
}
```

Newtyping Heads and Tails

Newtyping your data structures to give them semantics can prevent many subtle bugs.

```
type utail = u16;
type udefault = u32;

type AtomicTail = AtomicU16;
type AtomicHead = AtomicU32;

// Read and write permissions
struct RWHead<const C: usize>(*const AtomicHead);
struct RWTail<const C: usize>(*const AtomicTail);

// Read-only permission
struct ReadOnlyHead<const C: usize>(*const AtomicHead);
struct ReadOnlyTail<const C: usize>(*const AtomicTail);
```

Incorporating Newtypes Into Data Structure

Good newtypes communicate intent *clearly*.

```
pub struct Consumer<...> {
    tails: [RWTail<C>; T],
    heads: [ReadOnlyHead<C>; T],
    buffer: ReadOnlyBuffer<T, S, L>,
}

pub struct Producer<...> {
    pub head: RWHead<C>,
    pub tail: ReadOnlyTail<C>,
    pub buffer: RWBuffer<L>,
}
```

Const Generics Help With Safety

```
impl<
    const T: usize,    // # of producers
    const C: usize,    // bitsize of queue
    const S: usize,    // # of bytes (total)
    const L: usize,    // # of bytes (per producer)
    A: ThreadSafeAlloc, // custom allocator type
    > ProducerHandle<T, C, S, L, A> {
        // ...
    }
```

Reading From Queue With RAII

```rust
fn pop(&self, pid: usize) -> Vec<u8>;
```

Reading From Queue With RAII

```
fn pop(&self, pid: usize) -> Vec<u8>;

fn pop(&self, pid: usize, dst: &mut [u8]) -> usize;
```

Reading From Queue With RAII

```
fn pop(&self, pid: usize) -> Vec<u8>;

fn pop(&self, pid: usize, dst: &mut [u8]) -> usize;

fn pop<'a>(&'a mut self, pid: usize) -> &'a [u8];
```

Reading From Queue With RAII

```rust
fn pop(&self, pid: usize) -> Vec<u8>;

fn pop(&self, pid: usize, dst: &mut [u8]) -> usize;

fn pop<'a>(&'a mut self, pid: usize) -> &'a [u8];

fn pop<'a>(&'a mut self, pid: usize) -> Section<'a>;

struct Section<'a>{buffer: &'a [u8], ... };

impl<'a> Drop for Section<'a> {
    fn drop(&mut self) {
        unsafe {
            // increment tail atomically
        }
    }
}
```

Reading From Queue With RAII

```
// max capacity is 2^3 - 1
let (tx, mut rx) = wfmpsc::queue!(bitsize: 3, producers: 1);
tx[0].push(b"5678901");
{
    let mut section = rx.pop(0);
    for c in section.get_buffer().iter() {
        // iterate over section and do things
    }
} // dropping buffer
```

Reading From Queue With RAII

```
// max capacity is 2^3 - 1
let (tx, mut rx) = wfmpsc::queue!(bitsize: 3, producers: 1);
tx[0].push(b"5678901");
{
    let mut section = rx.pop(0);
    for c in section.get_buffer().iter() {
        // iterate over section and do things
    }
    let mut another_one = rx.pop(0);
    //                    ^^^^^^^^^^
    //                    |
    //                    + can't create another section
    //                      while previous one in scope
    black_box(&section);
} // dropping buffer
```

Runtime Analysis with Miri

What Is Miri?

Miri[6] is an intepreter for Rust's Mid-Level IR that dynamically checks for undefined behavior.

Checks include:

- OOB memory access & use-after-free
- Illegal memory alignments
- Reading from uninitialized memory
- Data races
- Violation of stacked borrows aliasing model

[6] https://github.com/rust-lang/miri

Issue #1: Uninitialized Arrays

Can you spot a potential problem here?

```
let mut producers: [Producer<...>; T] = { mem::zeroed() };

for (i, p) in producers.iter_mut().enumerate() {
        *p = self.get_producer_handle(i);
        //   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        //   |
        //   asssume this function returns a valid object
}
```

Issue #1: Uninitialized Arrays

Can you spot a potential problem here?

```
let mut producers: [Producer<...>; T] = { mem::zeroed() };

for (i, p) in producers.iter_mut().enumerate() {
        *p = self.get_producer_handle(i);
        //   ^^^^^^^^^^^^^^^^^^^^^^^^^^^
        //   |
        //   asssume this function returns a valid object
}
```

Problem: The assignment calls `Drop::drop` on the old value. This violates the producer's atomic refcount invariant.

Issue #1: Uninitialized Arrays

```
let mut producers: [MaybeUninit<Producer<...>>; T] =
    unsafe { MaybeUninit::uninit().assume_init() };

for (i, p) in producers.iter_mut().enumerate() {
    p.write(prod_handle(ptr, i as u8));
}
// FIXME: Cannot do mem::transmute from MaybeUninit to
// a const generic array.
// See https://github.com/rust-lang/rust/issues/61956
let prod_ptr = addr_of!(producers) as *const _;
let producers = unsafe { core::ptr::read(prod_ptr) };
```

Issue #2: Dangling Pointer

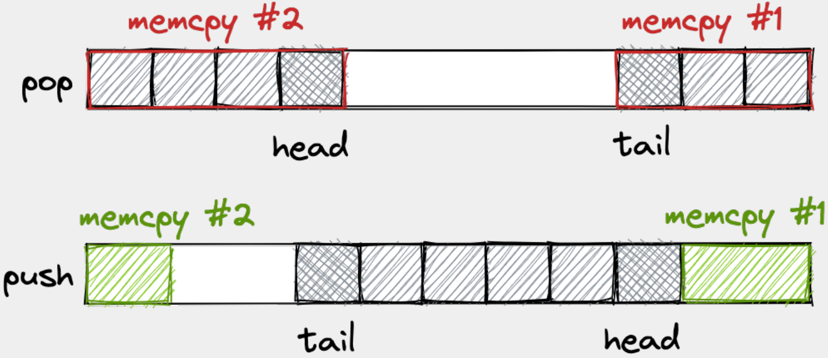


Figure: Elements can spill over the boundary of the ring buffer, so we need to invoke `memcpy` twice.

Issue #2: Dangling Pointer

```
// first memcpy
core::ptr::copy_nonoverlapping(
    src as *const u8,
    dst as *mut u8,
    L - head,
);
// second memcpy
core::ptr::copy_nonoverlapping(
    (src + C - head) as *const u8,
    self.buffer.0 as *mut u8,
    len - L + head,
);
```

Issue #2: Dangling Pointer

```
// first memcpy
core::ptr::copy_nonoverlapping(
    src as *const u8,
    dst as *mut u8,
    L - head,
);
// second memcpy
core::ptr::copy_nonoverlapping(
    (src + C - head) as *const u8,
    self.buffer.0 as *mut u8,
    len - L + head,
);
```

Issue #3: Incorrect Pointer Arithmetics (again)

```
error: unsupported operation: racy imperfectly overlapping
atomic access is not possible in the C++20 memory model,
and not supported by Miri's weak memory emulation
   --> /Users/zk/wfmpsc/src/lib.rs:275:13
    |
275 |             atomic.store(val, ord);
    |             ^^^^^^^^^^^^^^^^^^^^^^ racy imperfectly
    |             overlapping atomic access is not possible
    |             in the C++20 memory model, and not
    |             supported by Miri's weak memory emulation
```

Issue #3: Incorrect Pointer Arithmetics (again)

CONCLUSION

Conclusion

- Be cognisant of the language's semantic model

[7] https://doc.rust-lang.org/nomicon/

Conclusion

- Be cognisant of the language's semantic model
  - The Rustonomicon[7] is a good starting point

[7] https://doc.rust-lang.org/nomicon/

Conclusion

- Be cognisant of the language's semantic model
  - The Rustonomicon[7] is a good starting point
- Familiarize yourself with the memory models that underpin your stack

[7] https://doc.rust-lang.org/nomicon/

Conclusion

- Be cognisant of the language's semantic model
  - The Rustonomicon[7] is a good starting point
- Familiarize yourself with the memory models that underpin your stack
- Use RAII and lifetimes to create safe viewtypes

[7] https://doc.rust-lang.org/nomicon/

Conclusion

- Be cognisant of the language's semantic model
  - The Rustonomicon[7] is a good starting point
- Familiarize yourself with the memory models that underpin your stack
- Use RAII and lifetimes to create safe viewtypes
- Memory fragmentation is a powerful trade off

[7] https://doc.rust-lang.org/nomicon/

Conclusion

- Be cognisant of the language's semantic model
  - The Rustonomicon[7] is a good starting point
- Familiarize yourself with the memory models that underpin your stack
- Use RAII and lifetimes to create safe viewtypes
- Memory fragmentation is a powerful trade off
- Learn from the OGs

[7] https://doc.rust-lang.org/nomicon/

More Resources

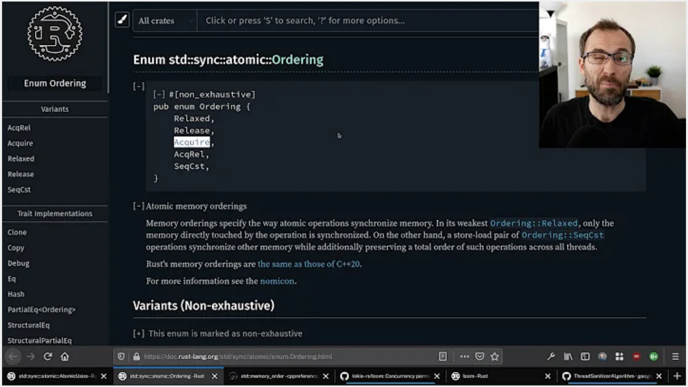


Figure: Atomics and Memory Ordering by Jon Gjengset [video]

Thanks for your attention!

References

Paul E McKenney.
Is parallel programming hard, and, if so, what can you do about it?
arXiv preprint arXiv:1701.00854, 2017.