Un-locked Zig

Synchronizing Code without Locks

Lukas Pietzschmann

October 25th, 2025

Motivation

- Have you ever cooked with others?
  - It's horrible!
- You need to coordinate who does what and when
  - Otherwise, you get in each other's way
- Same problem in programming

Generated by Google's Gemini

Concurrent Programming

```
fun Stack(comptime T: type) type {
    return struct {
        backing: std.ArrayList(T),

        pub fn push(self: *Stack, value: T) void {
            // try self.backing.append(value);
            const len = self.backing.items.len;
            self.backing.items.ptr[len] = value;
            self.backing.items.len = len + 1;
        }
    };
}
```

Concurrent Programming

```
fun Stack(comptime T: type) type {
    return struct {
        backing: std.ArrayList(T),
        lock: std.Thread.Mutex,

        pub fn push(self: *Stack, value: T) void {
            // try self.backing.append(value);
            self.lock.lock();
            const len = self.backing.items.len;
            self.backing.items.ptr[len] = value;
            self.backing.items.len = len + 1;
            self.lock.unlock();
        }
    };
}
```

Beyond Mutual Exclusion

- A Mutex is easy to understand and use
- Just grab it and you're safe!

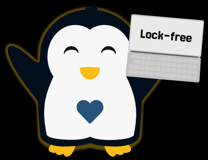
- But for more complex interactions, there are also more complex tools
  - Semaphores
  - Non-blocking Locks
  - Read-Write Locks
  - Reentrant Locks
  - Phases/Barriers
  - ...

The hidden Costs of Locks

Locks seem simple and safe

- But can easily create bottlenecks
- And add additional failure modes
- Easy to stop thinking about implications

What if we could achieve thread safety without ever forcing a thread to wait?



The hidden Costs of Locks

A selection of additional failure modes

- **Contention**
  Multiple threads try to acquire a lock leads to performance degradation.
- **Starvation**
  When many threads compete for a lock, some threads may never get it.
- **Priority Inversion**
  A lower-priority thread holds a lock needed by a higher-priority thread.
- **Composability**
  Locks don’t compose well, suggesting the addition of coarser-grained ones.

Agenda

What's up with Locks?

Lock-free Coding

Un-Locking Zig

Generated by Google's Gemini

How to synchronize without Locks?

The critical section should be so small that no other thread could interrupt it.

Our work horse: Compare and Swap (CAS)

```
fn cas(pointer: *T, expected: T, new: T) bool {
    if (pointer.* != expected) {
        return false;
    }
    pointer.* = new;
    return true;
}
```

“Look at this memory address. If it still contains the value I expected, then — and only then — update it to my new value.”

Building upon CAS

- We can now utilize CAS to actually set a value atomically

```
var current_val: T = atomic_load(prt);
var new_val: T = compute(current_val);
while (!cas(ptr, current_val, new_val)) {
    current_val = atomic_load(ptr);
    new_val = compute(current_val);
}
```

- Often, this logic is wrapped into atomic variables
  - They then provide atomic methods for getting, setting, and updating the value

Levels of Freedom

Obstruction Free

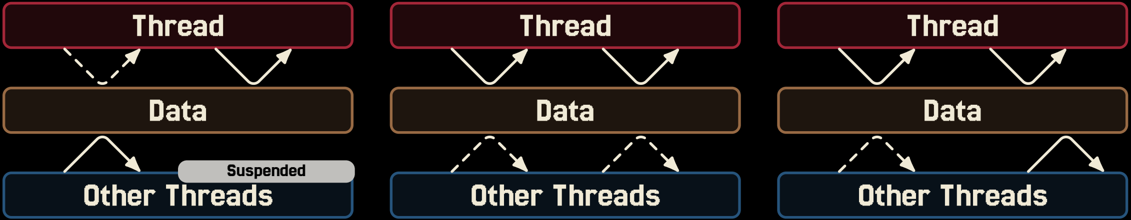
- Weekest Guarantee
- Thread will proceed if all other threads stop

Lock Free

- System-wide progress
- At least one thread makes progress

Wait Free

- Strongest Guarantee
- Per-thread progress



Levels of Freedom

Obstruction Free

- Weekest Guarantee
- Thread
  if all ot
  stop

Lock Free

- System-wide

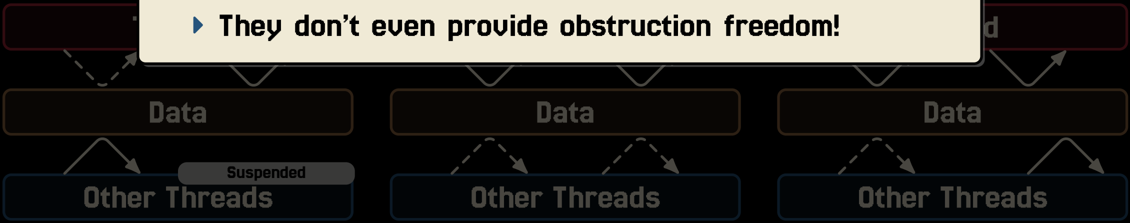
Wait Free

arantee

rogress

Blocking Algorithms

- Use locks or other blocking primitives
- A thread holding a lock can be suspended, blocking all other threads indefinitely
  - They don't even provide obstruction freedom!



Agenda

What's up with Locks?

Lock-free Coding

Un-Locking Zig

Un-Locking Zig — Atomic Operations

- Our CPUs support atomic operations withing their instruction sets
  - test-and-set, fetch-and-increment, compare-and-swap, ...
  - `LOCK XCHG`, `LOCK XADD`, `LOCK CMPXCHG`, ...
- Lock-free programming is enabled by the hardware itself

- Zig provides access to these atomic operations via built-in functions

```
@cmpxchgWeak(comptime T: type,
  ptr: *T,
  expected_value: T,
  new_value: T,
  success_order: AtomicOrder,
  fail_order: AtomicOrder
) ?T
```

```
@cmpxchgStrong(comptime T: type,
  ptr: *T,
  expected_value: T,
  new_value: T,
  success_order: AtomicOrder,
  fail_order: AtomicOrder
) ?T
```

Un-Locking Zig — Atomic Variables

- Zig provides atomic types in `std.atomic`
  - `std.atomic.Value`
- It provides wrappers around Zig's atomic built-ins

```
const std = @import("std");
const atomic = std.atomic;

var counter: atomic.Value(u32) = atomic.Value(u32).init(0);
counter.fetchAdd(1, .SeqCst);
```

Un-Locking Zig — Atomic Variables

- **Zig provides atomic types in** `std.atomic`
  - `std.atomic.Value`
- **It provides wrappers around Zig's atomic built-ins**

```
const std = @import("std");
const atomic = std.atomic;

var counter = 0;
@atomicRmw(u32, counter, .Add, 1, .SeqCst);
```

Let's look at some Code — Push

```
pub fn push(self: *Self, value: T) !void {
    var new_head = try self.allocator.create(Node);
    new_head.* = Node{
        .value = value,
        .next = null,
    };

    while (true) {
        const old_head = self.top.load(.acquire);
        new_head.next = old_head;
        if (self.top.cmpxchgWeak(old_head, new_head, .release, .acquire) == null) {
            return;
        }
    }
}
```

Let's look at some Code — Push

```
pub fn push(self: *Self, value: T) !void {
    var new_head = try self.allocator.create(Node);
    new_head.* = Node{
        .value = value,
        .next = null,
    };

    while (true) {
        const old_head = self.        (.acquire);
        new_head.next = old_he
        if (self.top.cmpxchgWe        ad, new_head, .release, .↩
     acquire) == null) {
            return;
        }
    }
}
```

Let's look at some Code — Pop

```
pub fn pop(self: *Self) ?T {
    while (true) {
        const old_head = self.top.load(.acquire) orelse {
            return null;
        };
        const new_head = old_head.next;
        if (self.top.cmpxchgWeak(old_head, new_head, .release, .acquire) == null) {
            const value = old_head.value;
            self.allocator.destroy(old_head);
            return value;
        }
    }
}
```

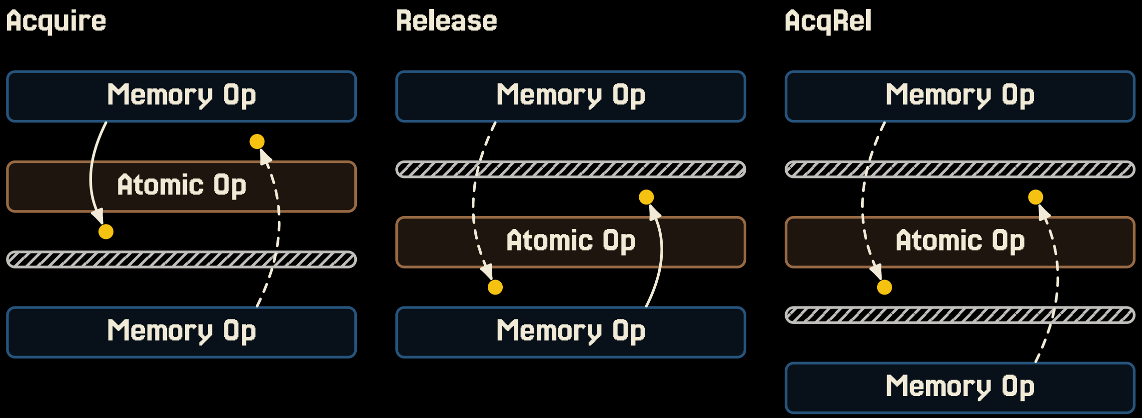
Why does CAS take so many parameters?

```
const data = 42;                    while (!data_ready) {}
const data_ready = true;            use(data);
```

- This seems to work, right?
  - Wrong! There is no clear dependency between `data_ready` and `data`
  - The compiler and the CPU might reorder instructions :)

- By introducing memory barriers, we can prevent this reordering
  - Since different operations have different requirements, we need to specify them individually

Bringing Order to Chaos



Bringing Order to Chaos

Acquire

Release

AcqRel

```
@cmpxchgWeak(comptime T: type,
  ptr: *T,
  expected_value: T,
  new_value: T,
  success_order: AtomicOrder,
  fail_order: AtomicOrder
) ?T
```

```
@cmpxchgStrong(comptime T: type,
  ptr: *T,
  expected_value: T,
  new_value: T,
  success_order: AtomicOrder,
  fail_order: AtomicOrder
) ?T
```

- **The success order is enforced when the the actual and expected values match**
- **Fail order is enforced when they don't**

Memory Op

The Problem with ~~ABBA~~ ABA

Generated by Google's Gemini

Generated by Google's Gemini

Generated by Google's Gemini

The Solution to ~~ABBA~~ ABA

DCAS

Double CAS; not supported on most hardware.

```
cas(ptr, ep, ap,
    ver, ev, av);
```

Not to be confused with a wide CAS!

Pointer Tagging

Only delay the problem, but can be practical.

On 8-byte aligned systems, 3 bits are free! We can just put the version there.

Then we don't need DCAS!

Hazard Pointers

Safe memory reclamation, but complex to implement.

Don't modify the CAS; prevent the "A back to A" part.

Pretty much manual garbage collection.

Agenda

What's up with Locks?

Lock-free Coding

Un-Locking Zig

Generated by Google's Gemini

When Locks are good enough

Simplicity

- When performance is not critical
- When few threads access a resource a few times

Coordination

- When threads need to wait for each other
- When complex interactions are needed

When Lock-Free shines

Performance

- Better performance under oversubscription
- Better suited for real-time and low-latency systems

Robustness

- No unpredictable blocking delays
- No deadlocks, livelocks, or priority inversions

Conclusion

- Locks have real — often hidden — costs
- Non-blocking algorithms utilize atomic operations to achieve thread safety
- A CAS inside a loop is the building block of many algorithms
- While non-blocking algorithms have actual advantages, they also come with their own challenges

> Don’t be afraid to use locks, but don’t limit yourself to them either!

LUKAS PIETZSCHMANN

Zigtoberfest, October 25th, 2025

lukas@pietzschmann.org