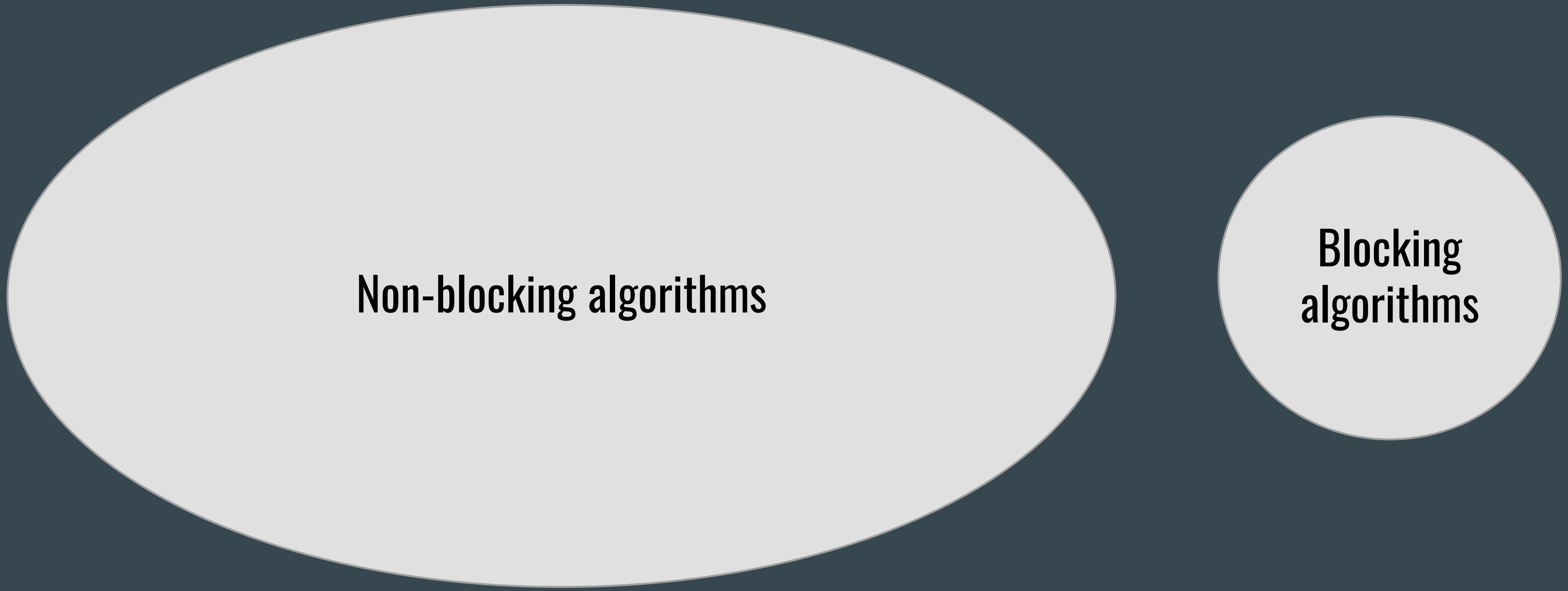
Non-blocking algorithms

Bruno Corrêa Zimmermann

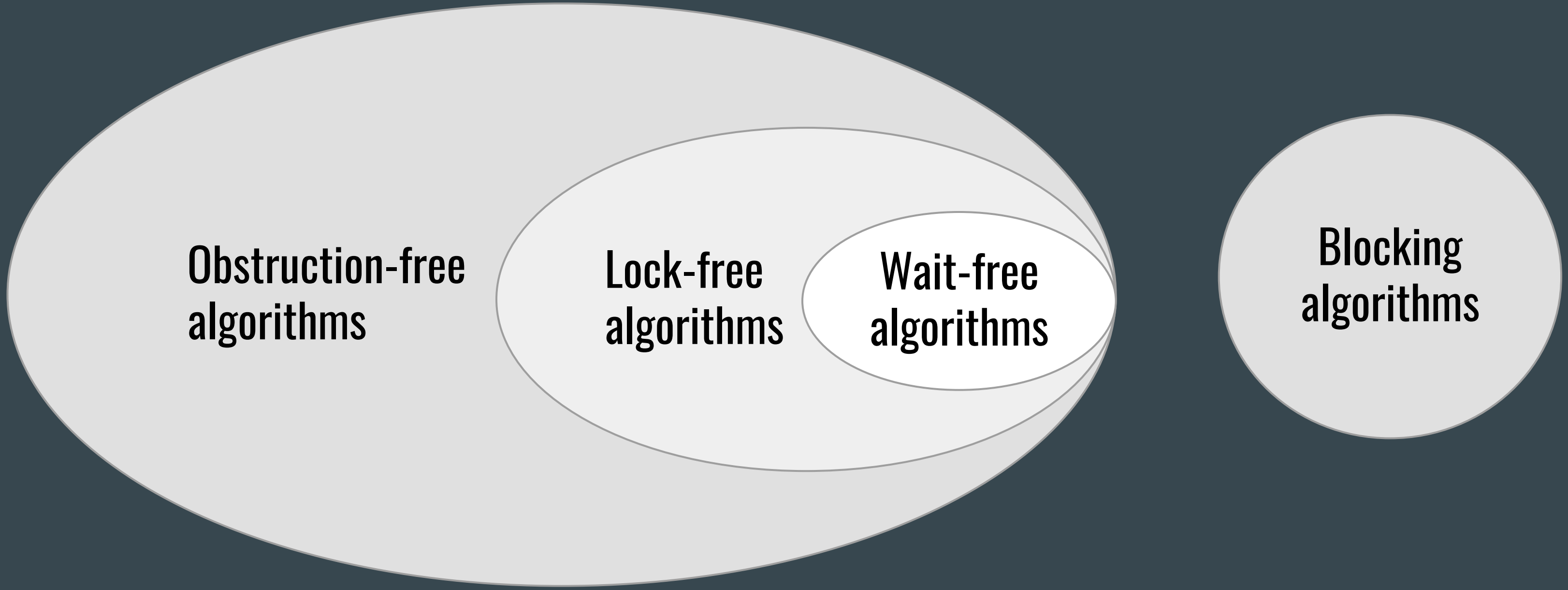
What is a non-blocking algorithm?

A concurrent algorithm that does not block the thread to synchronize.



What is a non-blocking algorithm?

Every non-blocking algorithm is at least an obstruction-free algorithm!



Non-blocking algorithm classes -comparison

Algorithm class	Invariant condition	Invariant
Obstruction-free	Suspend all threads except one	The remaining thread makes progress
Lock-free	Suspend one thread	At least one of the remaining threads makes progress
Wait-free	Suspend one thread	All remaining threads make progress

Why non-blocking algorithms?

- Guarantee that there won’t be any deadlocks;
- Progress even when other resources are busy;
- No need to depend on a scheduler;
- Possibly better performance.

Why NOT non-blocking algorithms?

- Easier to introduce bugs;
- Hard to implement with actual good performance;
- Your problem might not fit well into non-blocking algorithms.

How to synchronize without blocking?

- Bare read and writes to memory?
    - Data race: undefined behavior and data corruption;
    - Not an option!
- Use atomic memory operations instead:
    - Atomic operations’ side effects are observable only when finished.

Atomic memory operations

Atomic operation	Non-atomic version
`y = x.load();`	`y = *x;`
`x.store(y);`	`*x = y;`
`z = x.swap(y);`	`z = *x;` `*x = y;`

`z = x.compare_exchange(w, y);`	`x_ = *x;`
	`if x_ == w`
	`    *x = y;`
	`    z = Ok(x_);`
	`else`
	`    z = Err(x_);`

Memory access reordering

- Memory accesses can be reordered:
  - By the compiler;
  - By the processor.
- A thread cannot observe own operations' reorderings;
- A thread can observe other threads' reorderings;
- The programmer can restrict reorderings in atomic operations.

Memory orderings

- Memory orderings are types of restriction a programmer can put in reorderings.
- As of Rust 1.68.0, in Rust they are:
  - Sequential consistency (SeqCst);
  - Acquire;
  - Release;
  - Acquire/release (AcqRel);
  - Relaxed.

Memory orderings – sequential consistency and relaxed

- Sequential consistency = no reordering can cross this operation:
  - Worse performance but more easily correct;
- Relaxed = any reordering can cross this operation:
  - Better performance but more easily incorrect.

Memory orderings – sequential consistency

foo();

bar(); Compiler/processor can freely reorder inside this.

baz();

let y = x.load(SeqCst); ⟵ Compiler/processor cannot cross this.

bla();

blor(); Compiler/processor can freely reorder inside this.

blergh();

Memory orderings – relaxed

foo();

bar();

baz();

let y = x.load(Relaxed);

bla();

blor();

blergh();

Compiler/processor can freely reorder anywhere here

Memory orderings – acquire, release and acquire/release

- Acquire should be used for reads;
- Release should be used for writes;
- Acquire/release should be used for combining read and write in one operation;
- Acquire and release are paired together;
- Acquire/release is paired with acquire, release or acquire/release.

Memory orderings – acquire, release and acquire/release

- Acquire = operations before the associated write stays before the write;
- Release = operations after the associated read stays after the read;
- Acquire/release = the effects of an acquire and a release at the same time.

Memory orderings – acquire and release

Thread A

```
foo();
bar();
let z = x.load(Acquire);
use_a(a);
```

Thread B

```
a = 5;
x.store(y, Release);
blergh();
hogh();
```

— CAN be observed by the other thread

X CANNOT be observed by the other thread

Memory orderings – acquire/release

Thread A

foo();

bar();

let z = x.load(Acquire);

use_a(a);

Thread B

b = 5;

x.store(y, Release);

blergh();

hogh();

Thread C

borg();

a = 7;

let v = x.swap(w, AcqRel);

use_b(b);

sourgh();

Memory orderings – acquire/release – thread B and C

Thread A

foo();

bar();

let z = x.load(Acquire);

use_a(a);

Thread B

b = 5;

x.store(y, Release);

blergh();

hogh();

Thread C

borg();

a = 7;

let v = x.swap(w, AcqRel);

use_b(b);

sourgh();

— CAN be observed by the other thread

X CANNOT be observed by the other thread

Memory orderings – acquire/release – thread A and C

Thread A

```
foo();
bar();
let z = x.load(Acquire);
use_a(a);
```

Thread B

```
b = 5;
x.store(y, Release);
blergh();
hogh();
```

Thread C

```
borg();
a = 7;
let v = x.swap(w, AcqRel);
use_b(b);
sourgh();
```

— CAN be observed by the other thread

X CANNOT be observed by the other thread

Atomic data types in Rust standard library

- AtomicBool
- AtomicPtr<T>
- AtomicUsize
- AtomicIsize
- AtomicU8
- AtomicI8
- AtomicU16
- AtomicI16
- AtomicU32
- AtomicI32
- AtomicU64
- AtomicI64

Common atomic operations in Rust standard library

- ```
  fn load(&self, Ordering) -> T;
  ```
- ```
  fn store(&self, data: T, Ordering);
  ```
- ```
  fn swap(&self, data: T, Ordering) -> T;
  ```
- ```
  fn compare_exchange(
      &self,
      expected_value: T,
      new_value: T,
      success_ordering: Ordering,
      failure_ordering: Ordering,
  ) -> Result<T, T>;
  ```

Non-blocking algorithm tips

- Generally involves operations with reads and writes;
- Publish data atomically considering the implementation of consumers;
- Read data only when fully published;
- Cannot make a thread “wait” as if they were locks;
- Cannot use locks at all (mutex, read-write-locks, etc);
- Not even barriers.

Example: atomic, lock-free in-place factorial

```
use std::sync::atomic::{AtomicU64, Ordering::*};

pub fn update_to_factorial(number: &AtomicU64) {
    let mut current = number.load(SeqCst);
    loop {
        let factorial = (1 ..= current).product();
        match number.compare_exchange(current, factorial, SeqCst, Relaxed) {
            Ok(_) => break,
            Err(new) => current = new,
        }
    }
}
```

In the in-place factorial example...

- Usage of `compare_exchange`;
- Result is only published when fully done.

Counterexample: not a lockfree algorithm

```rust
use std::sync::atomic::{AtomicBool, Ordering::*};

struct Mutex {
    locked: AtomicBool,
}
impl Mutex {
    pub fn new() -> Self {
        Self { locked: AtomicBool::new(false) }
    }

    pub fn lock(&self) {
        while !self.locked.swap(true, Acquire) {}
    }
    pub fn unlock(&self) {
        self.locked.store(false, Release);
    }
}
```

In counterexample...

- It is actually a spinlock;
- `.lock()` will make the current thread wait:
  - possibly infinitely.

ABA Problem

- Arises designing some non-blocking algorithms;
- Affects `compare_exchange`;
- Mainly a issue with pointers.

ABA Problem

- Thread T reads pointer A;
- Thread U stores new pointer B;
- Thread U frees pointer A;
- Thread V reads pointer B;
- Thread V allocates new pointer;
  - Allocator recycles pointer A;
- Thread V stores recycled pointer A;
- Thread T compares-exchange expected A storing new pointer C;

ABA Problem

- Thread T succeeds:
  - Even though the pointer contents of A were different before recycling;
- Potential data corruption;
- This is the ABA problem:
  - Recycled pointers yielding successful comparisons;
- There's also a problem with freeing stuff other thread is reading.

ABA Problem – example – stack definition

```rust
use std::{alloc::{alloc, dealloc, Layout},
    ptr,
    sync::atomic::{AtomicPtr, Ordering::*}};

struct Node<T> {
    data: T,
    next: *mut Self,
}
pub struct Stack<T> {
    top: AtomicPtr<Node<T>>,
}
impl<T> Stack<T> {
    pub fn new() -> Self {
        Self { top: AtomicPtr::new(ptr::null_mut()) }
    }
}
impl<T> Drop for Stack<T> {
    fn drop(&mut self) { while let Some(_) = self.pop() {} }
}
```

ABA Problem – example – stack push

```
pub fn push(&self, data: T) {
    let mut top = self.top.load(Acquire);
    let node_ptr;
    unsafe {
        node_ptr = alloc(Layout::new::<Node<T>>()) as *mut Node<T>;
        *node_ptr = Node { data, next: top };
    }
    loop {
        match self.top.compare_exchange(top, node_ptr, Release, Acquire) {
            Ok(_) => break,
            Err(new_top) => {
                top = new_top;
                unsafe { (*node_ptr).next = top }
            }
        }
    }
}
```

ABA Problem – example – stack pop

```
pub fn pop(&self) -> Option<T> {
    let mut top = self.top.load(Acquire);
    loop {
        if top.is_null() {
             break None;
        }
        let next = unsafe { (*top).next };
        match self.top.compare_exchange(top, next, AcqRel, Acquire) {
            Ok(node_ptr) => unsafe {
                let data = ptr::read(&(*node_ptr).data);
                dealloc(node_ptr as *mut u8, Layout::new::<Node<T>>());
                break Some(data);
            }
            Err(new_top) => top = new_top,
        }
    }
}
```

ABA Problem – example – stack pop

```
pub fn pop(&self) -> Option<T> {
    let mut top = self.top.load(Acquire);
    loop {
        if top.is_null() {
             break None;
        }
        let next = unsafe { (*top).next };
        match self.top.compare_exchange(top, next, AcqRel, Acquire) {
            Ok(node_ptr) => unsafe {
                let data = ptr::read(&(*node_ptr).data);
                dealloc(node_ptr as *mut u8, Layout::new::<Node<T>>());
                break Some(data);
            }
            Err(new_top) => top = new_top,
        }
    }
}
```

Thread A: reads in pop()
Thread B: pops and frees A pointer
Thread B: pushes with new allocation
Thread B: pushes with recycled allocation A
Thread A: compares_exchange successfully
“next” likely changed
leading to corruption

How to solve ABA?

- Add a version tag to the pointer:
  - Reduces address size or can be architecture-dependent;
  - Does not solve the problem completely;
- Use “hazard pointers”:
  - Tricky to implement;
- Use the “incinerator”:
  - Performance decreases;
- Problem-specific solutions.

My solution to ABA – the Incinerator

- A struct consisting of:
  - An atomic counter of threads running critical sessions;
  - A list of pointers to be deallocated soon;
- When a thread wants to deallocate a pointer, check the counter:
  - if zero, then deallocate the pointer and the whole list;
  - if not zero, simply put the pointer in the list;
- When a thread is going to access a critical pointer, increment the counter:
  - When done, decrement the counter.

Possible Incinerator API – structs

```
pub struct Garbage {
    pub pointer: *mut u8,
    pub layout: Layout,
}

pub struct Incinerator { /* ... */ }

pub struct Pause<'incin> { /* ... */ }
```

Possible Incinerator internal data

```
struct GarbageNode {
    element: Garbage,
    next: *mut GarbageNode,
}

pub struct Incinerator {
    critical_counter: AtomicUsize,
    garbage_list: AtomicPtr<Vec<GarbageNode>>,
}

pub struct Pause<'incin> {
    incinerator: &'incin Incinerator,
}
```

Possible Incinerator API – methods

```
impl Incinerator {
    pub fn new() -> Self;
    pub unsafe fn incinerate(&self, garbage: Garbage) -> bool;
    pub fn try_clear(&self) -> bool;
    pub fn pause<'a>(&'a self) -> Pause<'a>;
}

unsafe impl Send for Incinerator {}
unsafe impl Sync for Incinerator {}

impl Drop for Incinerator { /* ... */ }
impl<'a> Drop for Pause<'a> { /* ... */ }
```

Fixing our stack – definition

```
pub struct Stack<T> {
    top: AtomicPtr<Node<T>>,
    incinerator: Arc<Incinerator>,
}

impl<T> Stack<T> {
    pub fn new() -> Self {
        Self::with_incinerator(Arc::new(Incinerator::new()))
     }

    pub fn with_incinerator(incinerator: Arc<Incinerator>) -> Self {
        Self { top: AtomicPtr::new(ptr::null_mut()), incinerator }
    }
}
```

Fixing our stack – pop

```
pub fn pop(&self) -> Option<T> {
    let _incinerator_guard = self.incinerator.pause();
    let mut top = self.top.load(Acquire);
    loop {
        if top.is_null() {
             break None;
        }
        let next = unsafe { (*top).next };
        match self.top.compare_exchange(top, next, AcqRel, Acquire) {
            Ok(node_ptr) => unsafe {
                let data = ptr::read(&(*node_ptr).data);
                self.incinerator.incinerate(Garbage {
                    pointer: node_ptr as *mut u8,
                    layout: Layout::new::<Node<T>>(),
                });
                break Some(data);
            }
            Err(new_top) => top = new_top,
        }
    }
}
```

Thank you!

Questions?