Paul E. McKenney, Meta Platforms Kernel Team

Kangrejos 2023, September 17, 2023

Lifetime-End Pointer Zap in Rust

© 2023 Meta Platforms

Overview

- Problem statement
- Current Rust practice
- Future directions?

Problem Statement

Problem Statement (C11, 1/2)

```
struct node_t* _Atomic top;

void list_push(value_t v)
{
 struct node_t *newnode = (struct node_t *) malloc(sizeof(*newnode));

 set_value(newnode, v);
 newnode->next = atomic_load(&top);
 do {
  // newnode->next may have become invalid
 } while (!atomic_compare_exchange_weak(&top, &newnode->next, newnode));
}
```

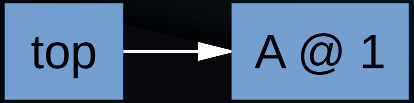
Problem Statement (C11, 2/2)

```
void list_pop_all()
{
  struct node_t *p = atomic_exchange(&top, NULL);

  while (p) {
    struct node_t *next = p->next;

    foo(p);
    free(p);
    p = next;
  }
}
```

Problem Illustration (C11)

Freelist

Initial State

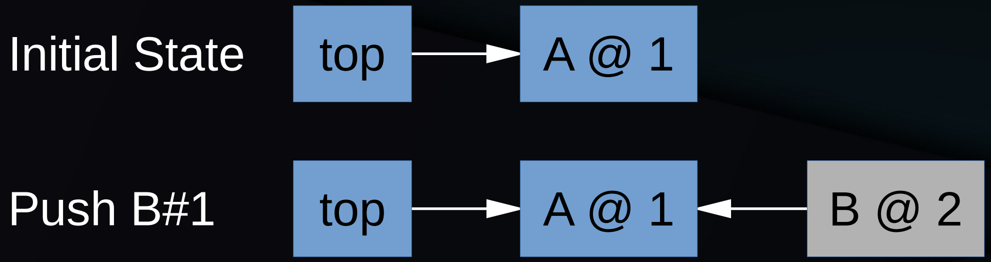


top

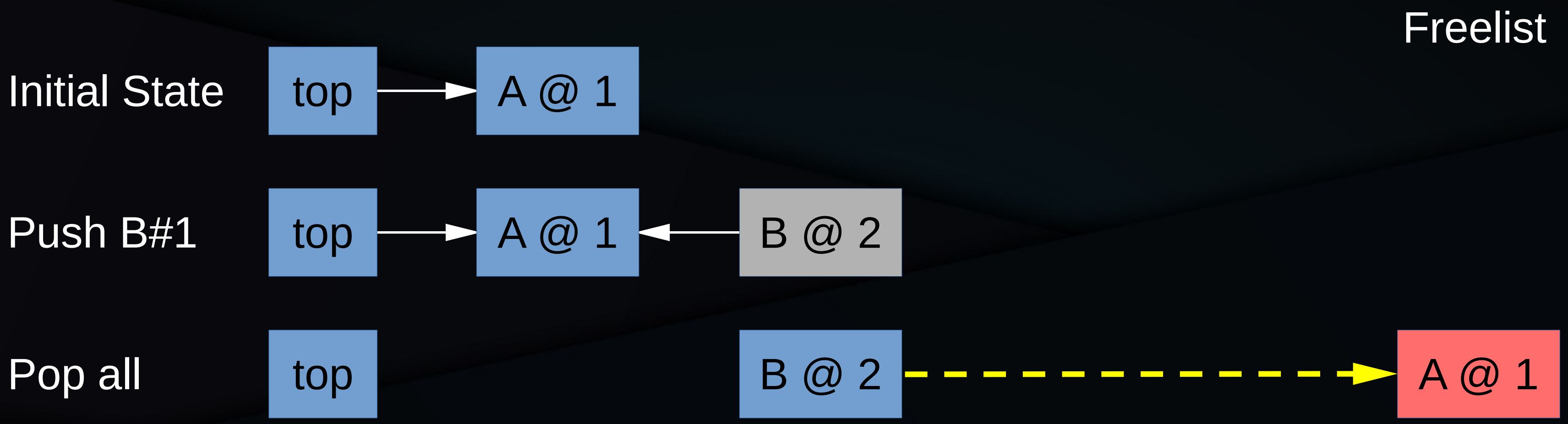
A @ 1

Problem Illustration (C11)

Freelist



Problem Illustration (C11)



Freelist

Initial State

top

A @ 1

Push B#1

top

A @ 1

B @ 2

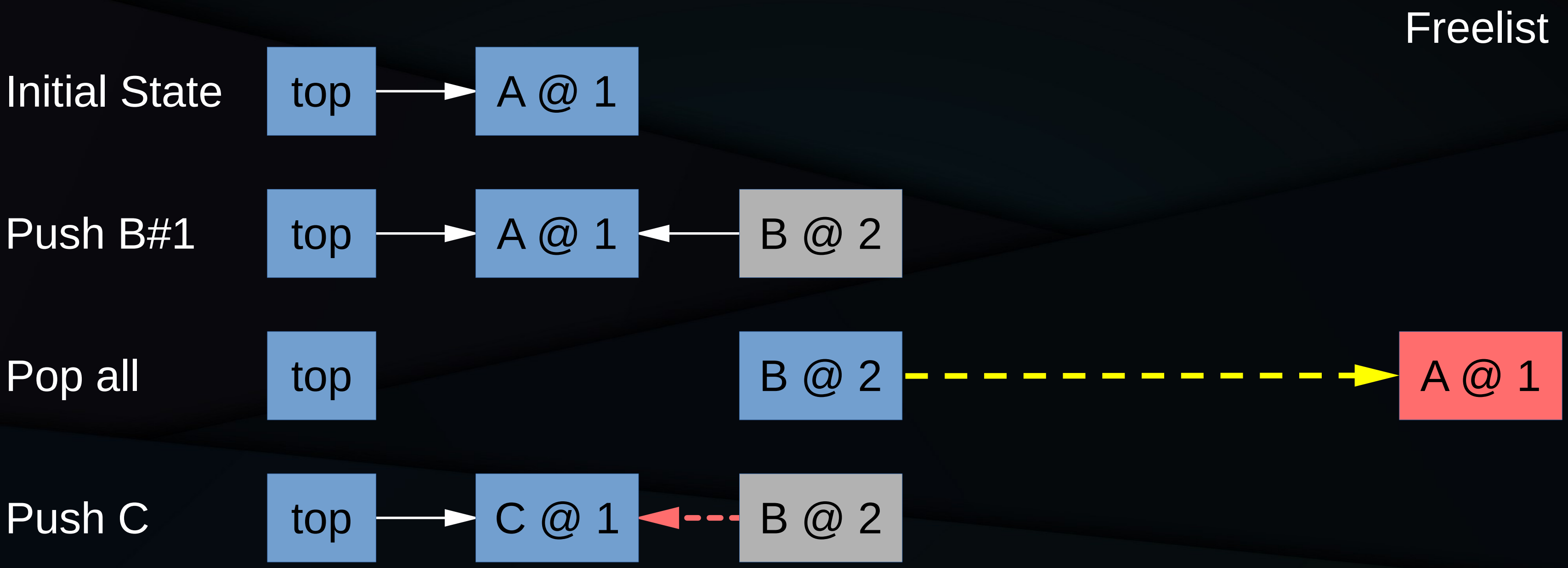
Pop all

top

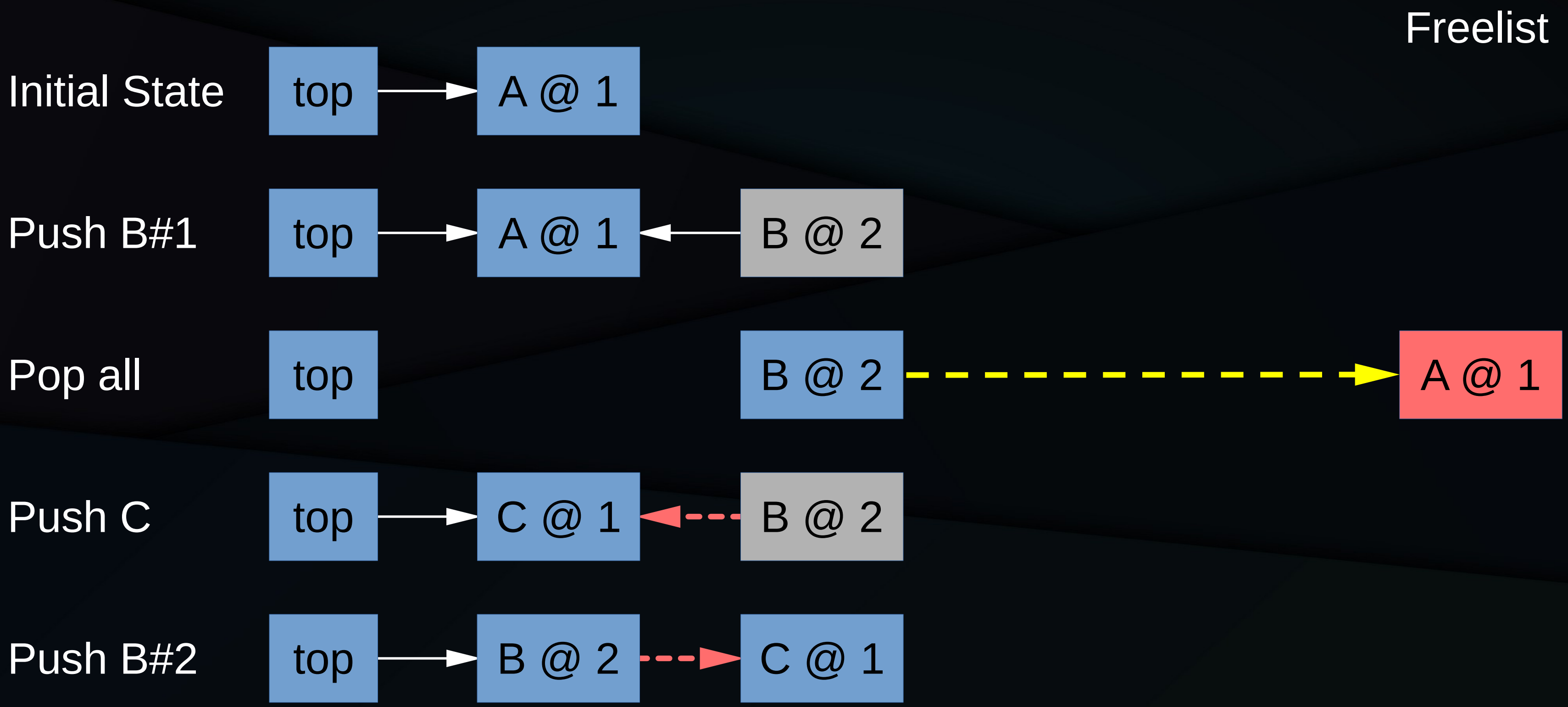
B @ 2

A @ 1

Problem Illustration (C11)



Problem Illustration (C11)



Freelist

Initial State: top → A @ 1

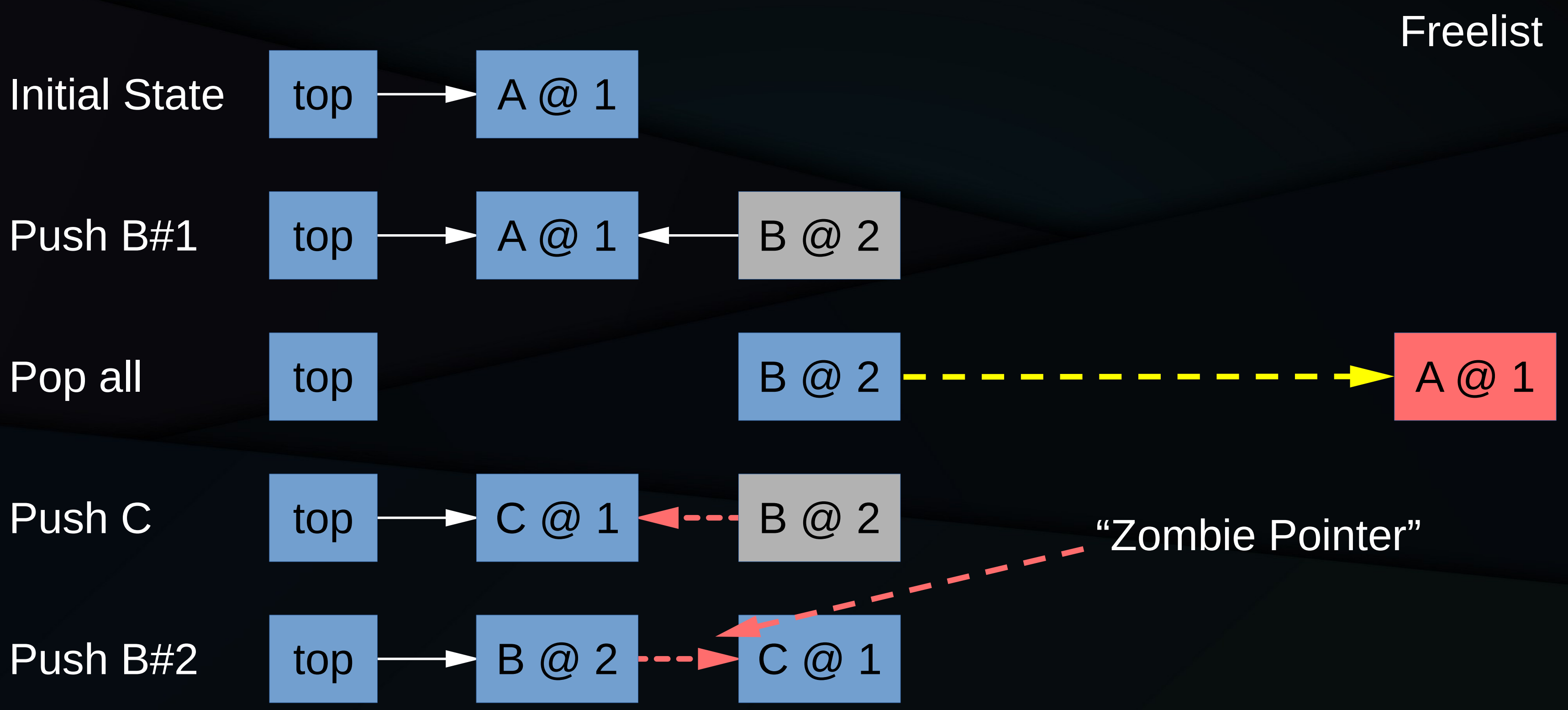
Push B#1: top → A @ 1 ← B @ 2

Pop all: top B @ 2 → A @ 1

Push C: top → C @ 1 ← B @ 2

Push B#2: top → B @ 2 → C @ 1

Problem Illustration (C11)



Freelist

Initial State

top

A @ 1

Push B#1

top

A @ 1

B @ 2

Pop all

top

B @ 2

A @ 1

Push C

top

C @ 1

B @ 2

“Zombie Pointer”

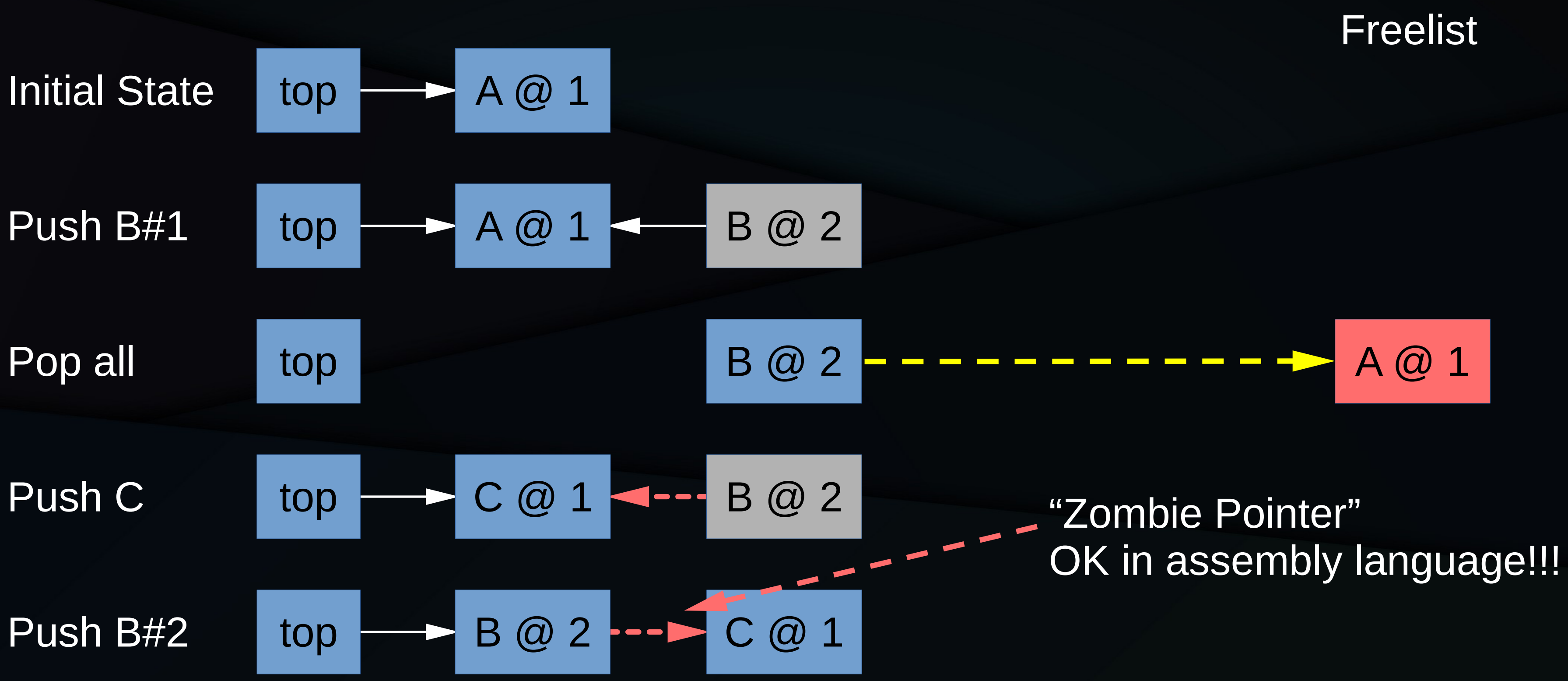
Push B#2

top

B @ 2

C @ 1

Problem Illustration (C11)



Freelist

Initial State top A @ 1

Push B#1 top A @ 1 B @ 2

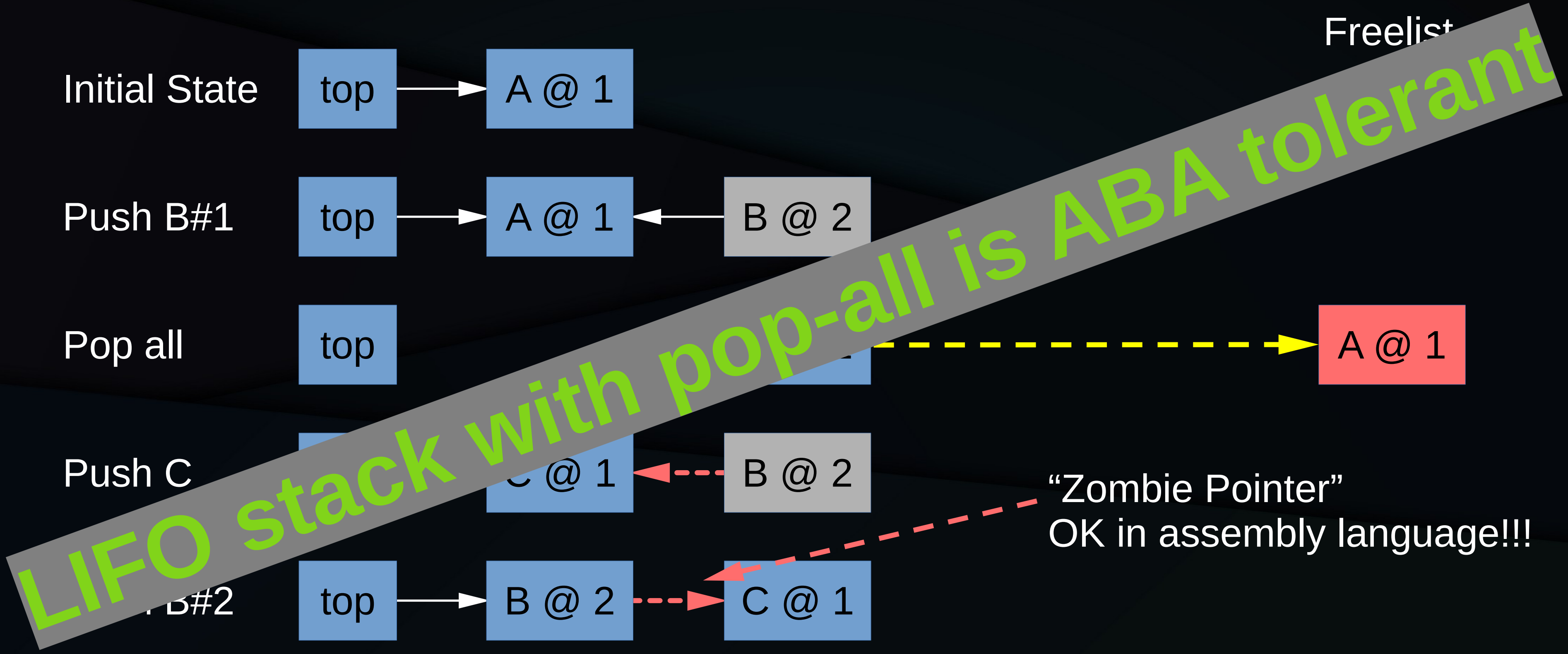
Pop all top B @ 2 A @ 1

Push C top C @ 1 B @ 2

Push B#2 top B @ 2 C @ 1

“Zombie Pointer”
OK in assembly language!!!

Problem Illustration (C11)



LIFO stack with pop-all is ABA tolerant

“Zombie Pointer”
OK in assembly language!!!

Why Worry About Novel Algorithms?

- LIFO stack described by Treiber in 1986
  - Written in IBM BAL, avoiding issues with compilers
- LIFO stack alluded to in early 1970s
- LIFO stack implemented in Rust library
  - Though with `pop()`, not `pop_all()`.
- Hence, LIFO stack not at all novel

RCU Workaround (C11, 1/2)

```
struct node_t* _Atomic top;

void list_push(value_t v)
{
  struct node_t *newnode = (struct node_t *) malloc(sizeof(*newnode));

  set_value(newnode, v);
  rcu_read_lock();
  newnode->next = atomic_load(&top);
  do {
    // newnode->next may have become invalid
  } while (!atomic_compare_exchange_weak(&top, &newnode->next, newnode));
  rcu_read_unlock();
}
```

Problem Statement (C11, 2/2)

```
void list_pop_all()
{
  struct node_t *p = atomic_exchange(&top, NULL);

  while (p) {
    struct node_t *next = p->next;

    foo(p);
    kfree_rcu(p);
    p = next;
  }
}
```

Current Rust Practice

Current Rust Practice

- Rust LIFO `Stack<T>` uses `SharedIncin`
- A simple RCU-like mechanism
  - Hat tip to livejournal commenter 94.134.180.48
  - “Will Your Rust Code Survive the Attack of the Zombie Pointers?”
    - https://paulmck.livejournal.com/64730.html

Rust Workaround (1/2)

```
pub fn push(&self, val: T) {
    let mut target =
        OwnedAlloc::new(Node::new(val, self.top.load(Acquire)));

    loop {
        let new_top = target.raw().as_ptr();
        match self.top.compare_exchange(
            target.next, new_top, Release, Relaxed,) {
            Ok(_) => {
                target.into_raw();
                break;
            },
            Err(ptr) => target.next = ptr,
        }
    }
}
```

https://docs.rs/lockfree/latest/src/lockfree/stack.rs.html

Rust Workaround (2/2)

```
pub fn pop(&self) -> Option<T> {
    let pause = self.incin.inner.pause();
    let mut top = self.top.load(Acquire);

    loop {
        let mut nnptr = NonNull::new(top)?;
        match self.top.compare_exchange(
            top, unsafe { nnptr.as_ref().next },
            AcqRel, Acquire,) {
            Ok(_) => {
                let val = unsafe { (&mut *nnptr.as_mut().val as *mut T).read() };
                pause.add_to_incin(unsafe { OwnedAlloc::from_raw(nnptr) });
                break Some(val);
            },
            Err(new_top) => top = new_top,
        }
    }
}
```

https://docs.rs/lockfree/latest/src/lockfree/stack.rs.html

Rust Workaround (2/2)

```
pub fn pop(&self) -> Option<T> {
    let pause = self.incin.inner.pause();
    let mut top = self.top.load(Acquire);

    loop {
        let mut nnptr = NonNull::new(top)?;
        match self.top.compare_exchange(
            top, unsafe { nnptr.as_ref().next },
            AcqRel, Acquire,) {
            Ok(_) => {
                let val = unsafe { (&mut *nnptr.as_mut().val as *mut T).read() };
                pause.add_to_incin(unsafe { OwnedAlloc::from_raw(nnptr) });
                break Some(val);
            },
            Err(new_top) => top = new_top,
        }
    }
}
```

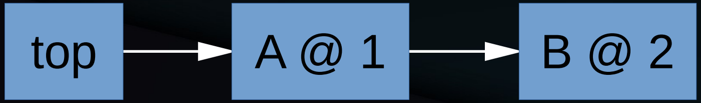
Deferred free, a form of RCU

Non-Problem Push Illustration (Rust)

Non-Problem Push Illustration (Rust)

Incinerator

Initial State

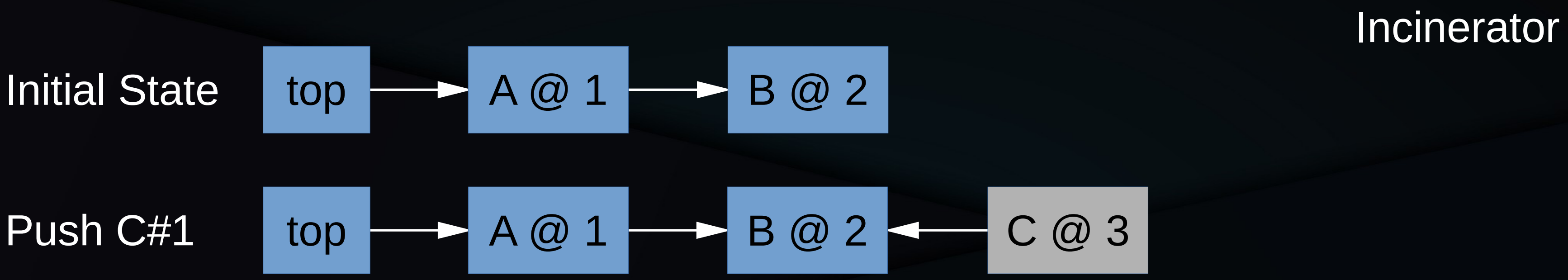


top

A @ 1

B @ 2

Non-Problem Push Illustration (Rust)



Incinerator

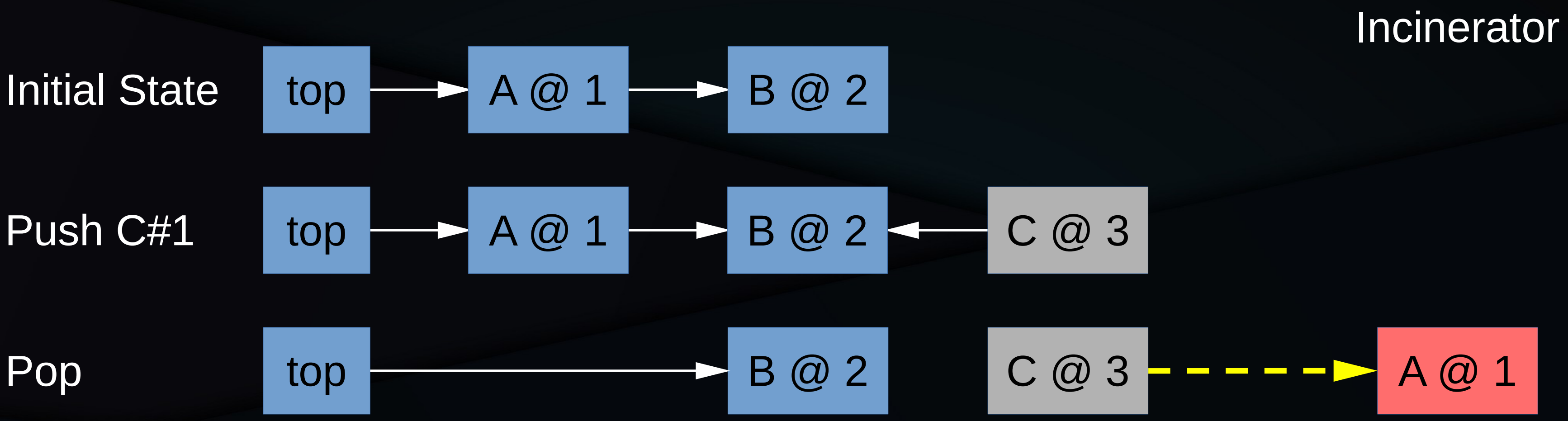
Initial State

top → A @ 1 → B @ 2

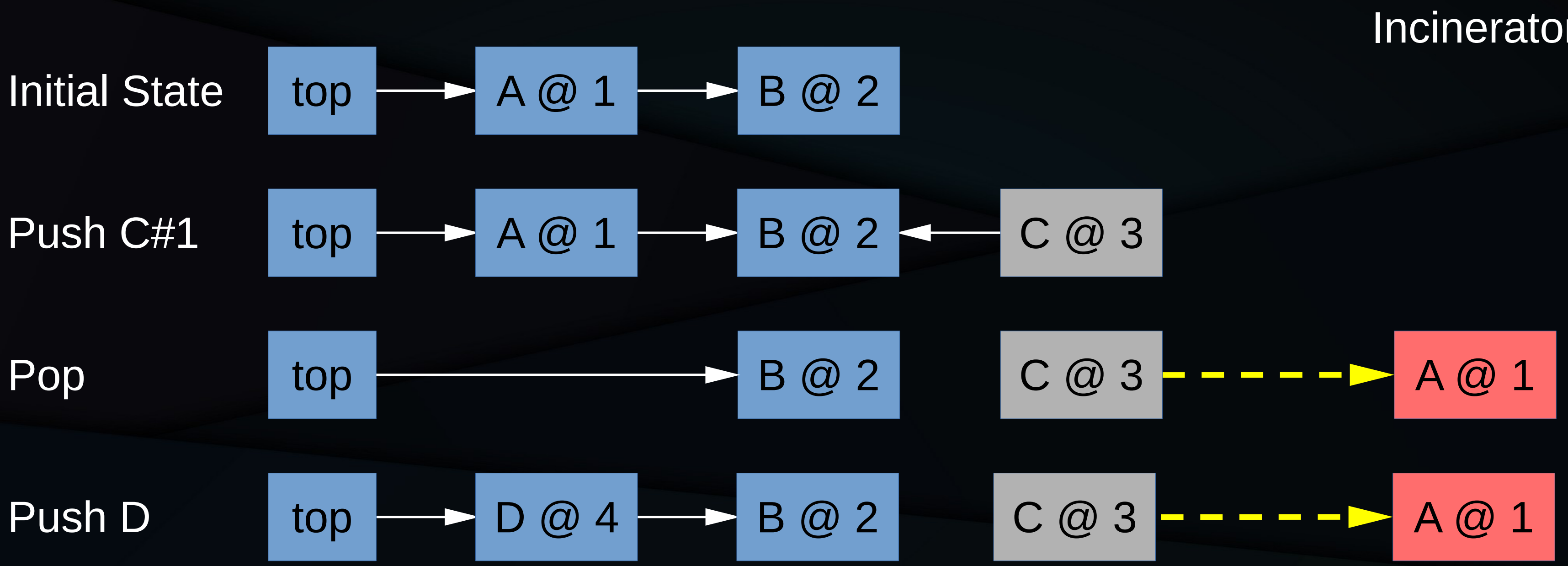
Push C#1

top → A @ 1 → B @ 2 ← C @ 3

Non-Problem Push Illustration (Rust)



Non-Problem Push Illustration (Rust)



Incinerator

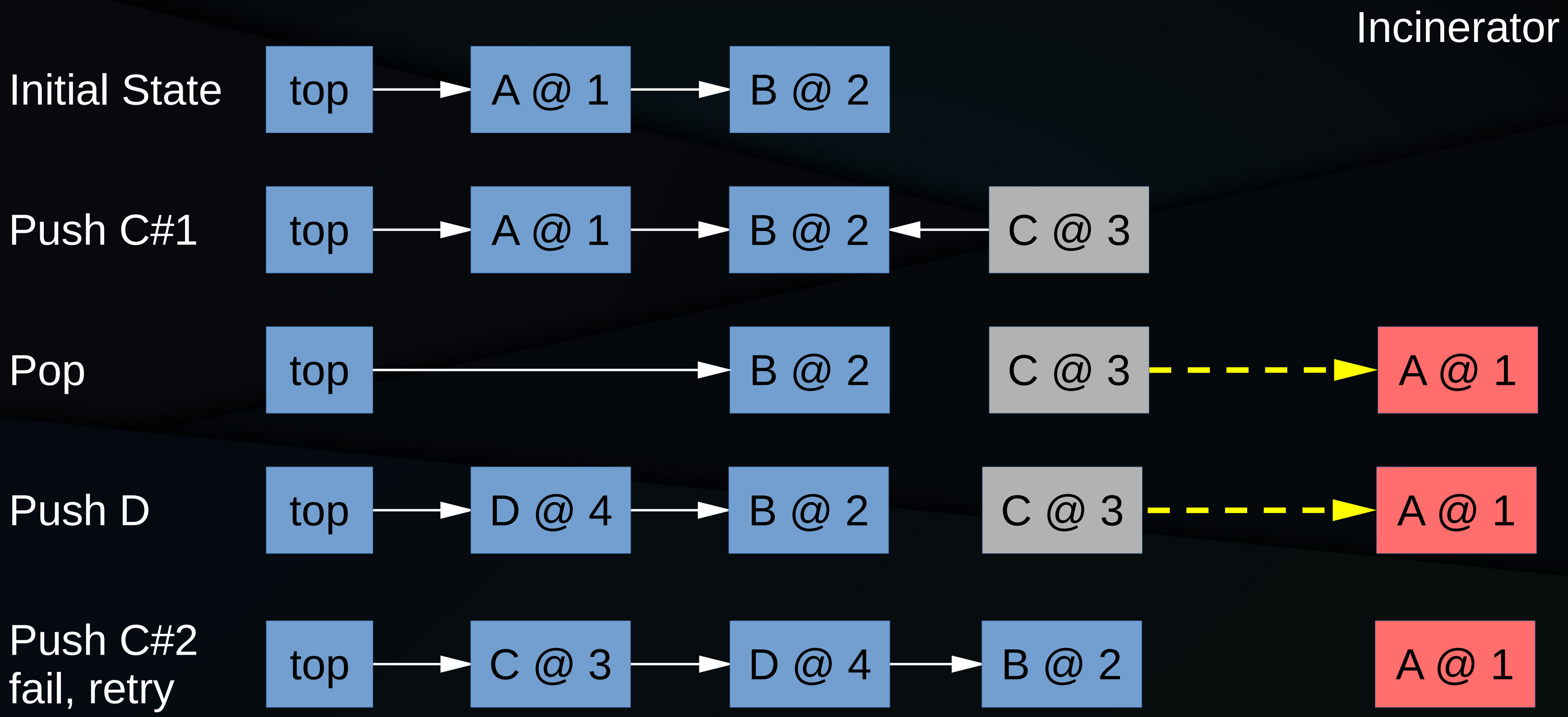
Initial State: top → A @ 1 → B @ 2

Push C#1: top → A @ 1 → B @ 2 ← C @ 3

Pop: top → B @ 2 C @ 3 → A @ 1

Push D: top → D @ 4 → B @ 2 C @ 3 → A @ 1

Non-Problem Push Illustration (Rust)



Incinerator

Initial State: top → A @ 1 → B @ 2

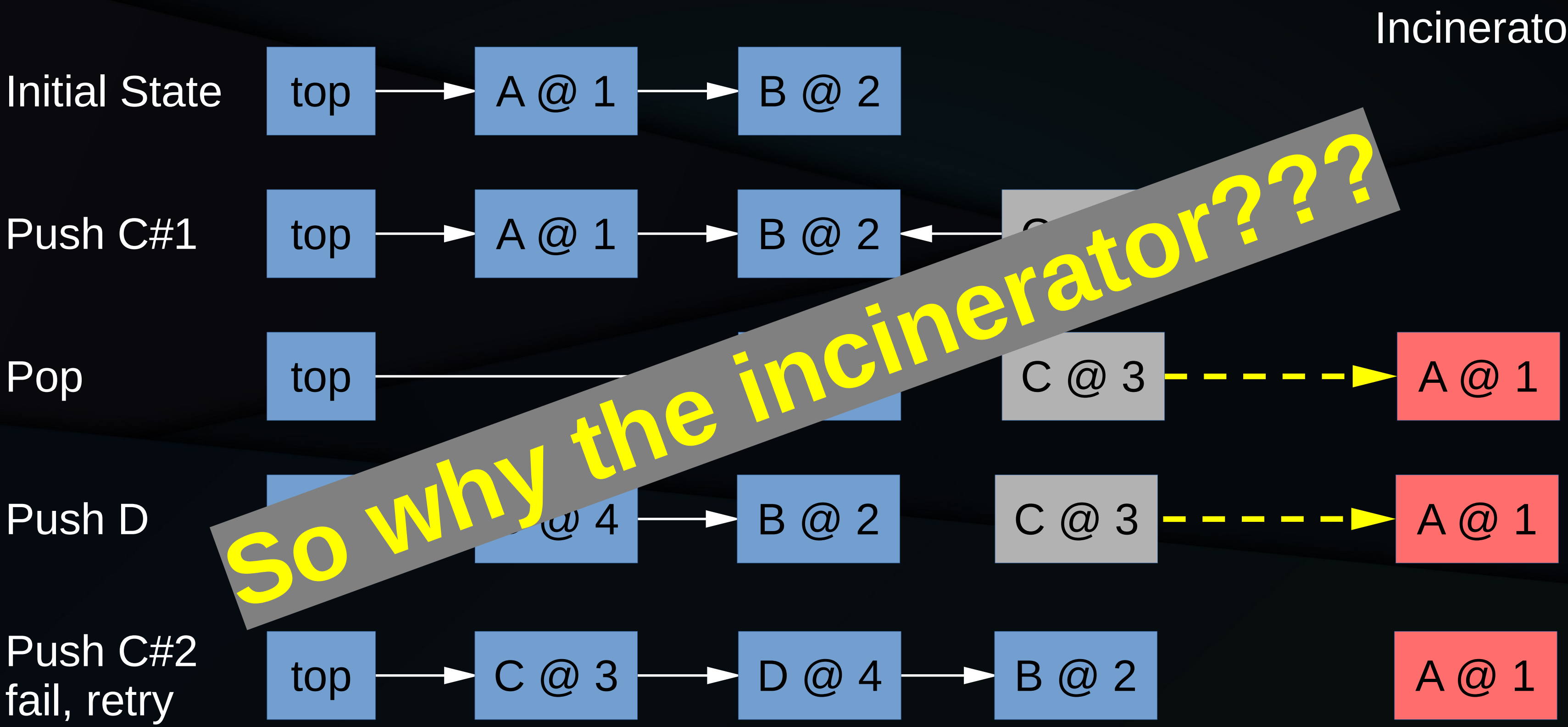
Push C#1: top → A @ 1 → B @ 2 ← C @ 3

Pop: top → B @ 2 C @ 3 → A @ 1

Push D: top → D @ 4 → B @ 2 C @ 3 → A @ 1

Push C#2 fail, retry: top → C @ 3 → D @ 4 → B @ 2 A @ 1

Non-Problem Push Illustration (Rust)



Incinerator

Initial State: top → A @ 1 → B @ 2

Push C#1: top → A @ 1 → B @ 2 ← C

Pop: top C @ 3 - - → A @ 1

Push D: @ 4 → B @ 2 C @ 3 - - → A @ 1

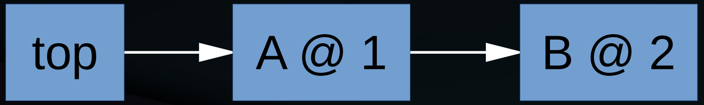
Push C#2 fail, retry: top → C @ 3 → D @ 4 → B @ 2 A @ 1

So why the incinerator???

Problem Pop Illustration (Rust-ish)

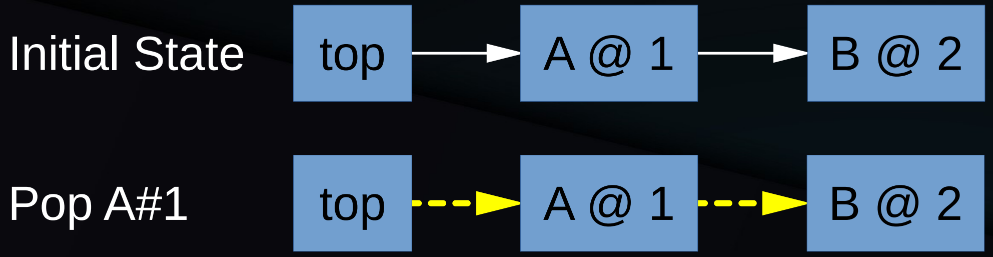
Problem Pop Illustration (Rust-ish)

Initial State



~~Incinerator~~
Freelist

Problem Pop Illustration (Rust-ish)



Initial State

top

A @ 1

B @ 2

Pop A#1

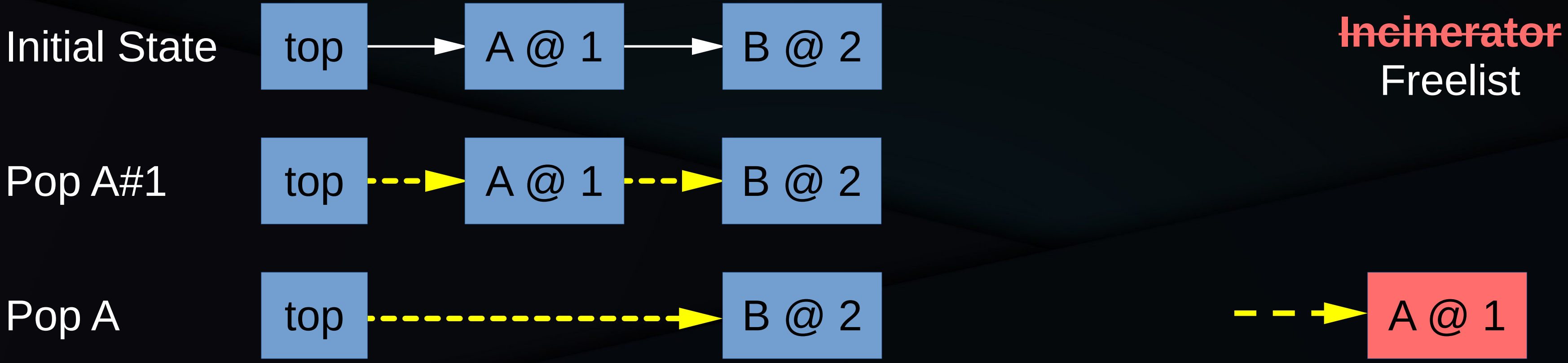
top

A @ 1

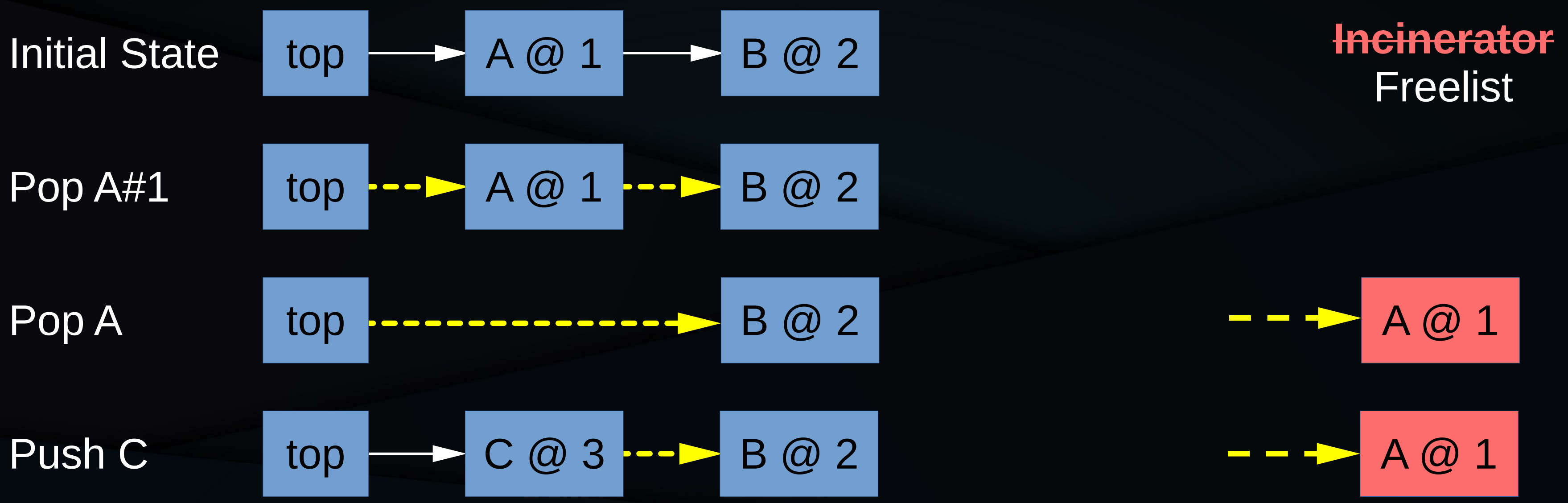
B @ 2

~~Incinerator~~
Freelist

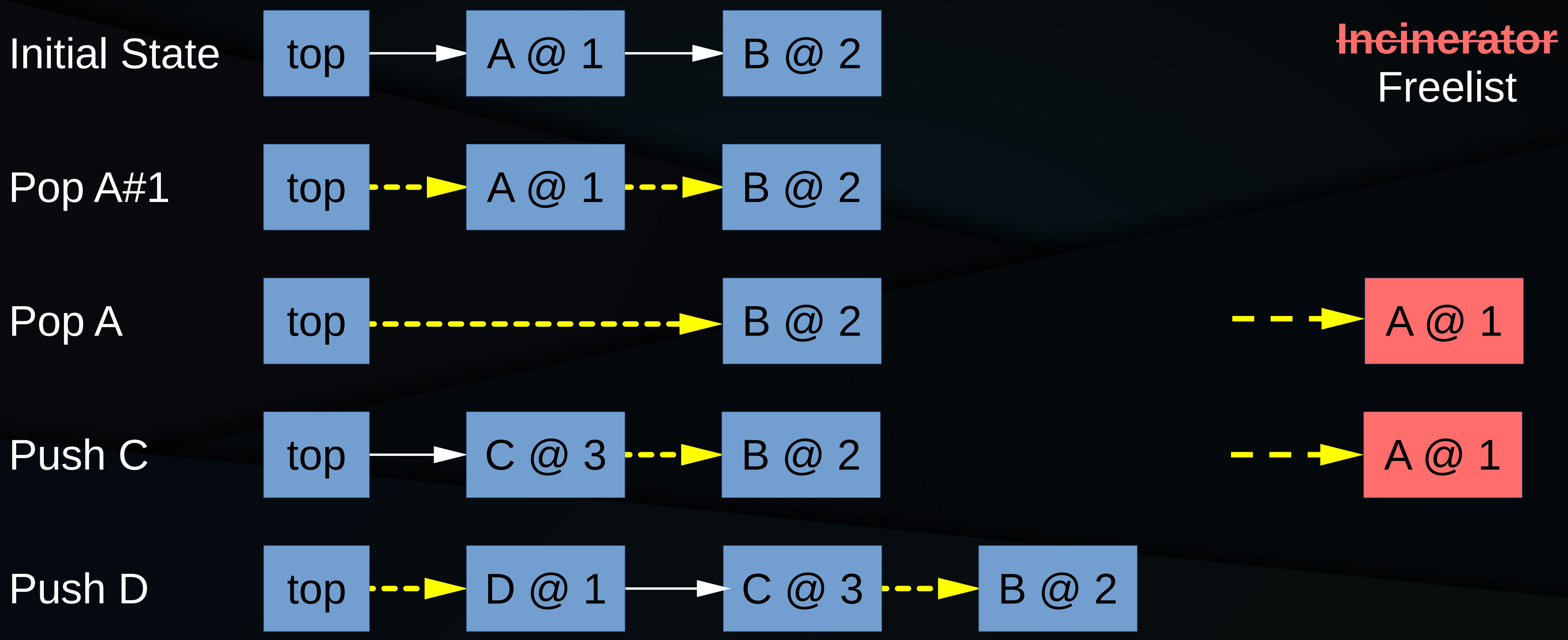
Problem Pop Illustration (Rust-ish)



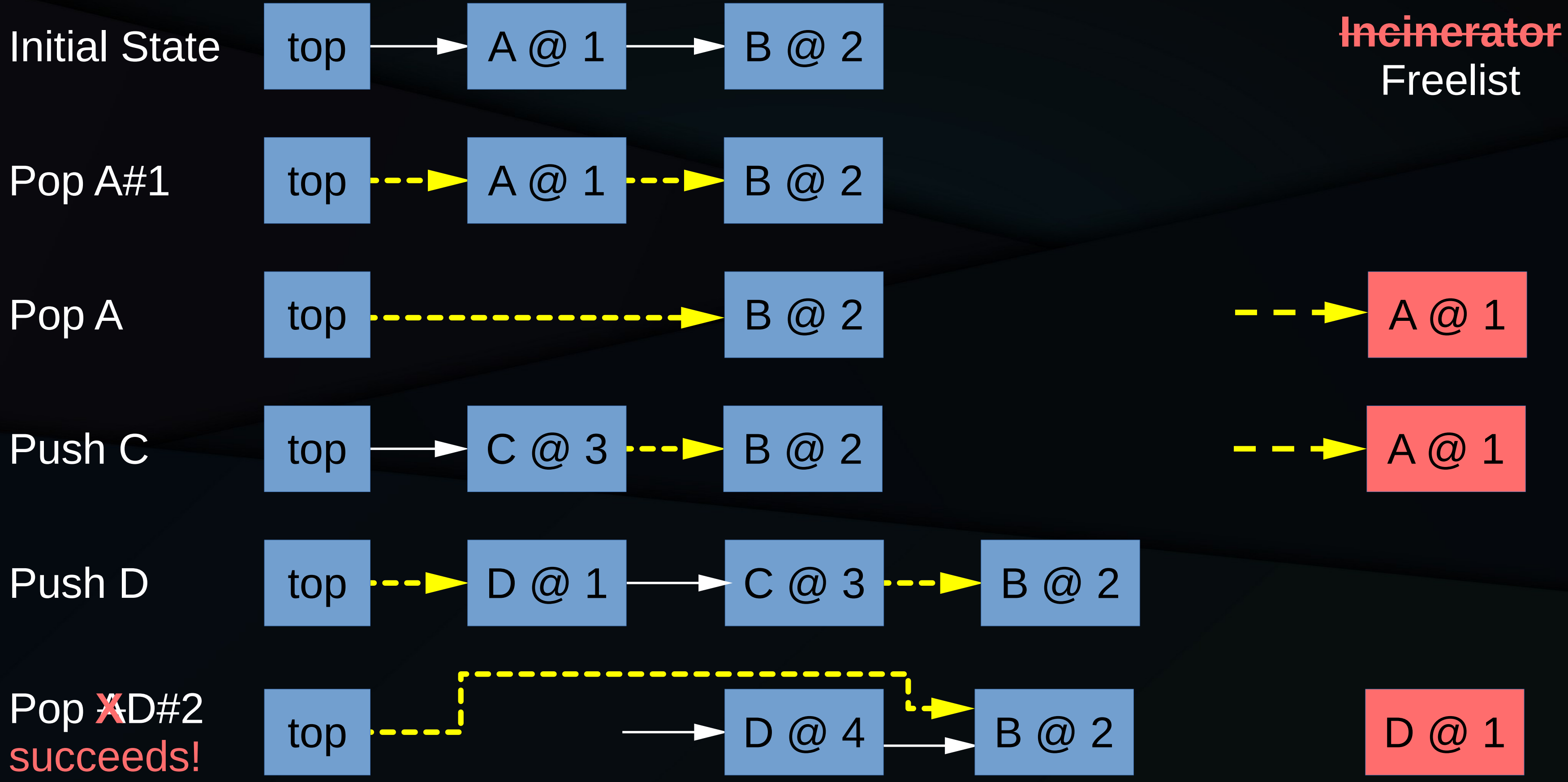
Problem Pop Illustration (Rust-ish)



Problem Pop Illustration (Rust-ish)



Problem Pop Illustration (Rust-ish)



Initial State: top → A @ 1 → B @ 2

~~Incinerator~~
Freelist

Pop A#1: top → A @ 1 → B @ 2

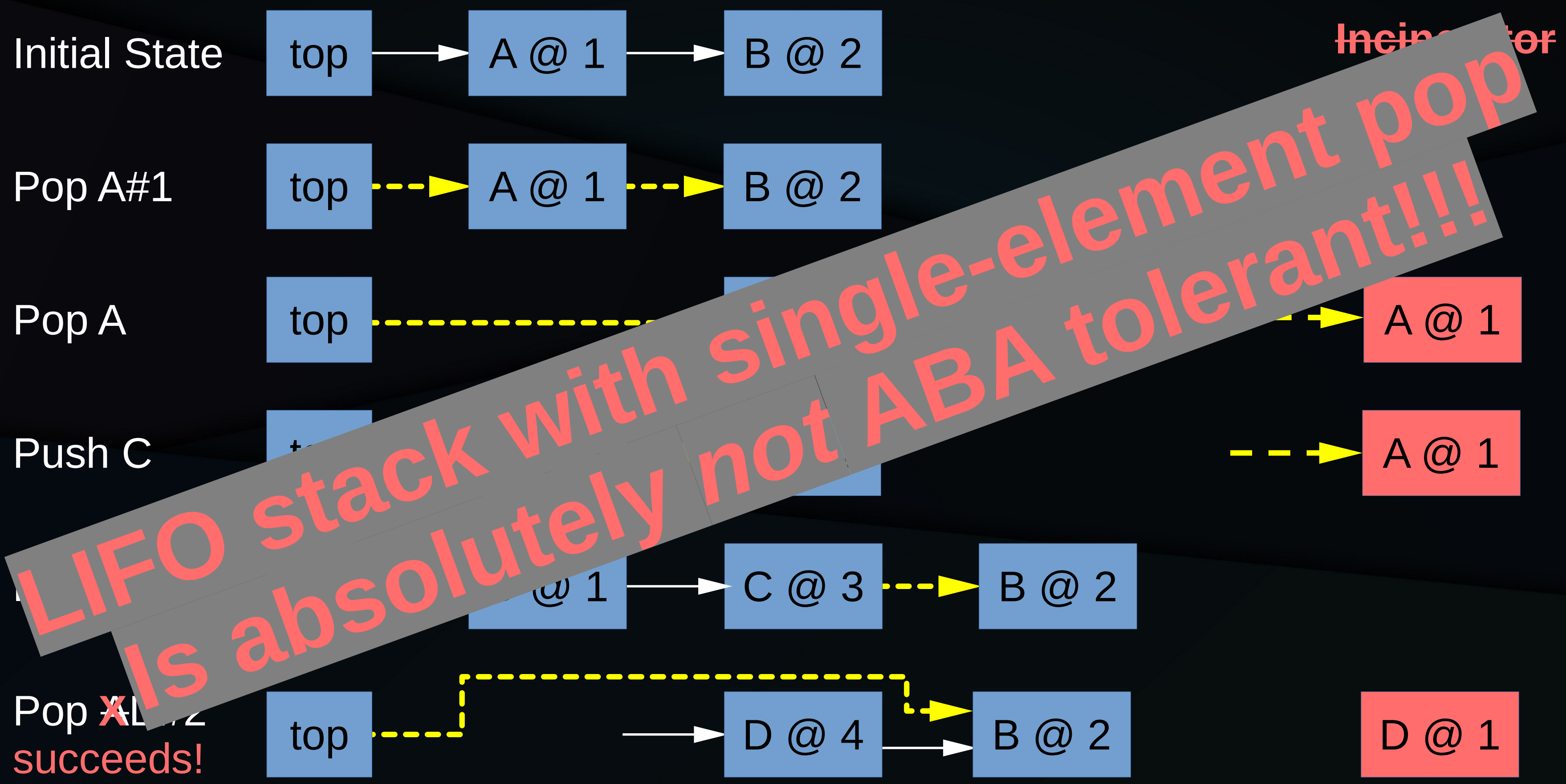
Pop A: top → B @ 2; → A @ 1

Push C: top → C @ 3 → B @ 2; → A @ 1

Push D: top → D @ 1 → C @ 3 → B @ 2

Pop ~~A~~D#2
succeeds!
top → B @ 2; → D @ 4 → B @ 2; D @ 1

Problem Pop Illustration (Rust-ish)



Initial State

Pop A#1

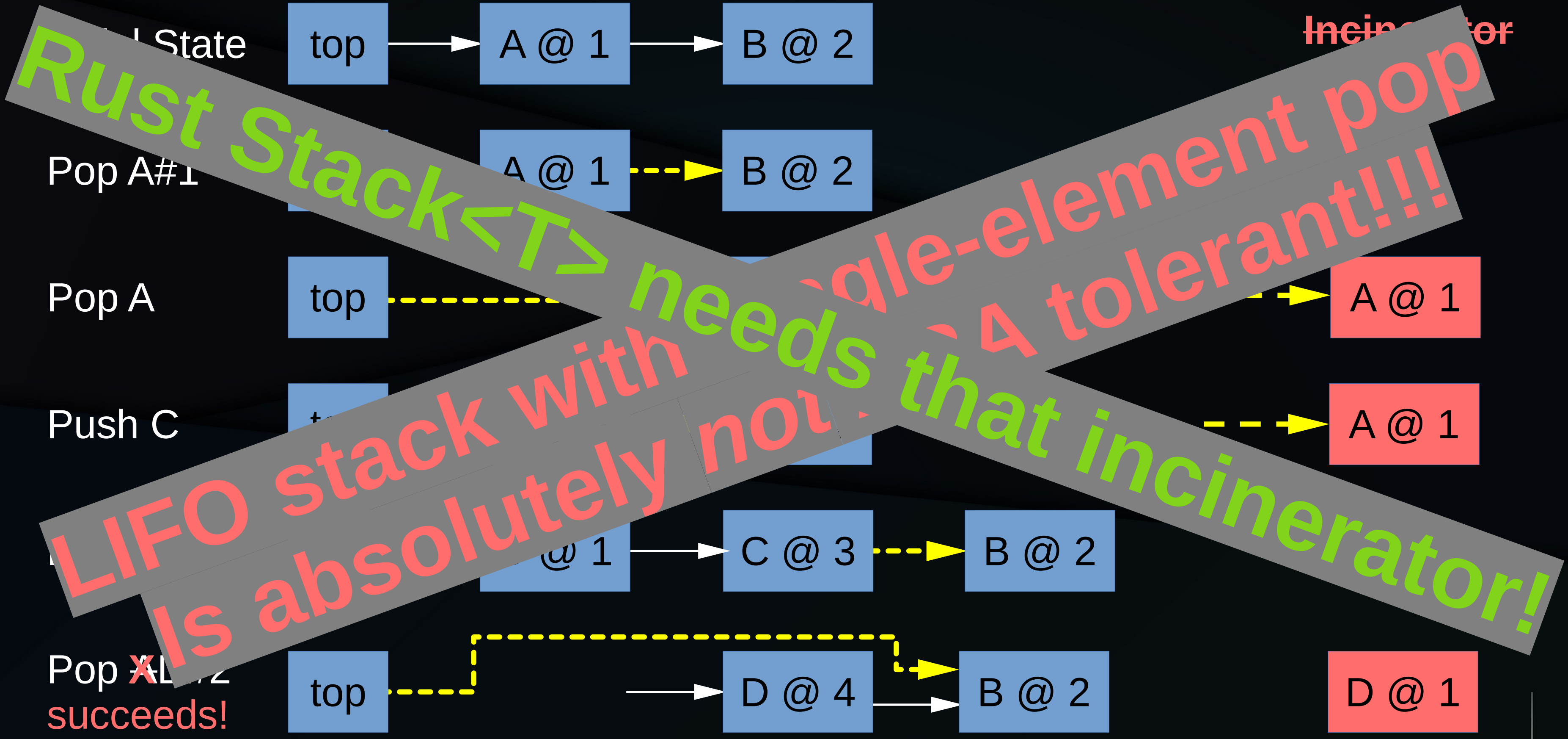
Pop A

Push C

Pop ~~A~~ X L…/2
succeeds!

LIFO stack with single-element pop
Is absolutely not ABA tolerant!!!

Problem Pop Illustration (Rust-ish)



Initial State

top

A @ 1

B @ 2

~~Incinerator~~

Pop A#1

A @ 1

B @ 2

Pop A

top

A @ 1

Push C

A @ 1

C @ 3

B @ 2

Pop ~~A~~ D#2

succeeds!

top

D @ 4

B @ 2

D @ 1

Rust Stack<T> needs that incinerator!

LIFO stack with single-element pop

Is absolutely not ABA tolerant!!!

Future Directions?

So What Is The Problem???

- Just defer free in both C, C++, and Rust!!!
- But this has costs if only pop-all is used:
  - Otherwise pointless deferred-free mechanism
  - Increased memory footprint
  - Increased CPU overhead
- Plus there are other use cases...

Why `push()` and `pop_all()`???

- “Server thread” use case
- Client threads `push()` requests
- Server thread does `pop_all()` and handles all requests up to that point
- This use case is often performance-critical and can appear in memory-constrained environments

Other Uses of Invalid Pointers

- Optimized sharded locks
- Hazard-pointer `try_protect()`
- Checking `realloc()` return value (Rust?)
- Pointers as keys and identity-only pointers
- Weak pointers (Android)

P1726R5: Pointer lifetime-end zap (informational/historical) https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p1726r5.pdf

Other Uses of Invalid Pointers

- Optimized sharded locks
- Hazard-pointer `try_protect()`
- Checking `realloc()` r value (Rust?)
- Pointers as l and identity-only pointers
- W ointers (Android)

Most need only stable comparisons

P1726R5: Pointer lifetime-end zap (informational/historical) https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p1726r5.pdf

How To Solve This Problem?

- Avoid using ABA-tolerant algorithms
  - Or pretend that such algorithms are not ABA-tolerant
  - Either way, Just Say No
    - For example, defer freeing of memory (as Rust `Stack<T>` does)
- Hide the memory allocator from the compiler
  - Attractive in standalone applications with special memory allocators
- Provide means to tell compiler to recompute provenance
  - Atomics, volatiles, and marking pointers safe (recursively)

Recompute Provenance!!!

- Recompute provenance on pointers:
  - Affected by atomic operations, including old pointer in successful CAS
  - Affected by volatile operation
  - Passed through `recompute_provenance()`
    - Including pointers reached via the returned pointer
- Non-comparison non-dereference computations involving invalid pointers must use representation bytes
  - Including normal loads and stores

Recompute Provenance Key Points

- Volatile operations require this anyway
  - Rust device driver interacting with Rust firmware!!!
- Nothing is lost in atomics, as they change behind the compiler’s back anyway, and by design
- Nothing is lost via `recompute_provenance()` because compiler cannot invent pointer comparisons

More Exciting Proposed Solution

- Anthony Williams:
  - P2188R1: Zap the Zap: Pointers are sometimes just bags of bits
  - https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2188r1.html
- Quite popular, except with compiler writers

Summary

Summary

- There are performance-critical ABA-tolerant algorithms
- Deferred free can handle them, but at a cost
- But why not enable no-extra-cost implementation of ABA-tolerant algorithms?

For More Information

- C N2369: Pointer lifetime-end zap
  - https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2369.pdf
- C++ P1726R5: Pointer lifetime-end zap (informational/historical)
  - https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p1726r5.pdf
- CPPCON: Will Your Rust Code Survive the Attack of the Zombie Pointers?
  - https://paulmck.livejournal.com/64730.html
- Blog: Will Your Rust Code Survive the Attack of the Zombie Pointers?
  - https://paulmck.livejournal.com/64730.html
- C++ P2414R1: Pointer lifetime-end zap proposed solutions
  - https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2414r1.pdf
- C++ P2188R1: Zap the Zap: Pointers are sometimes just bags of bits
  - https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2188r1.html

Questions?