Threads

Aaron Turon

Systems programming without the hassle

Systems programming without the hassle

crashes

heisenbugs

fear

Parallel!
^Systems programming without the hassle

crashes

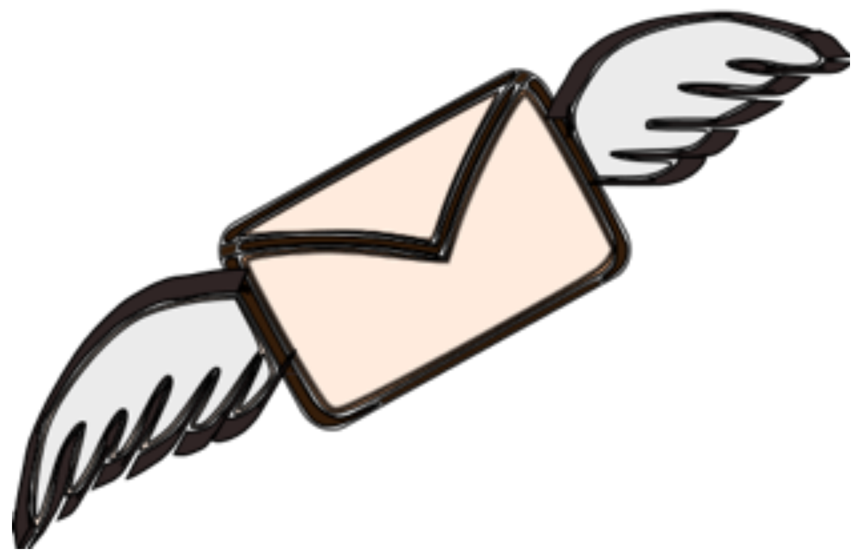
heisenbugs

fear

Concurrency and Parallelism

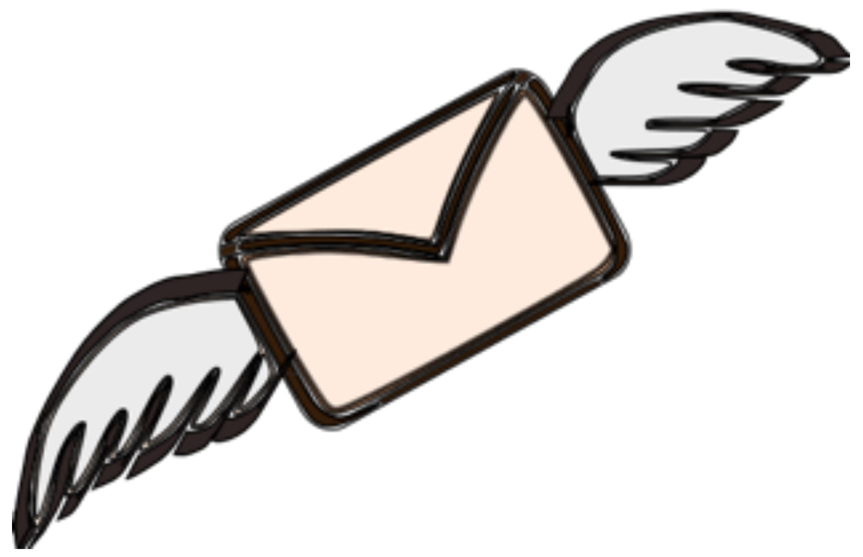
Multiparadigm

Multiparadigm



message-passing

Multiparadigm



message-passing

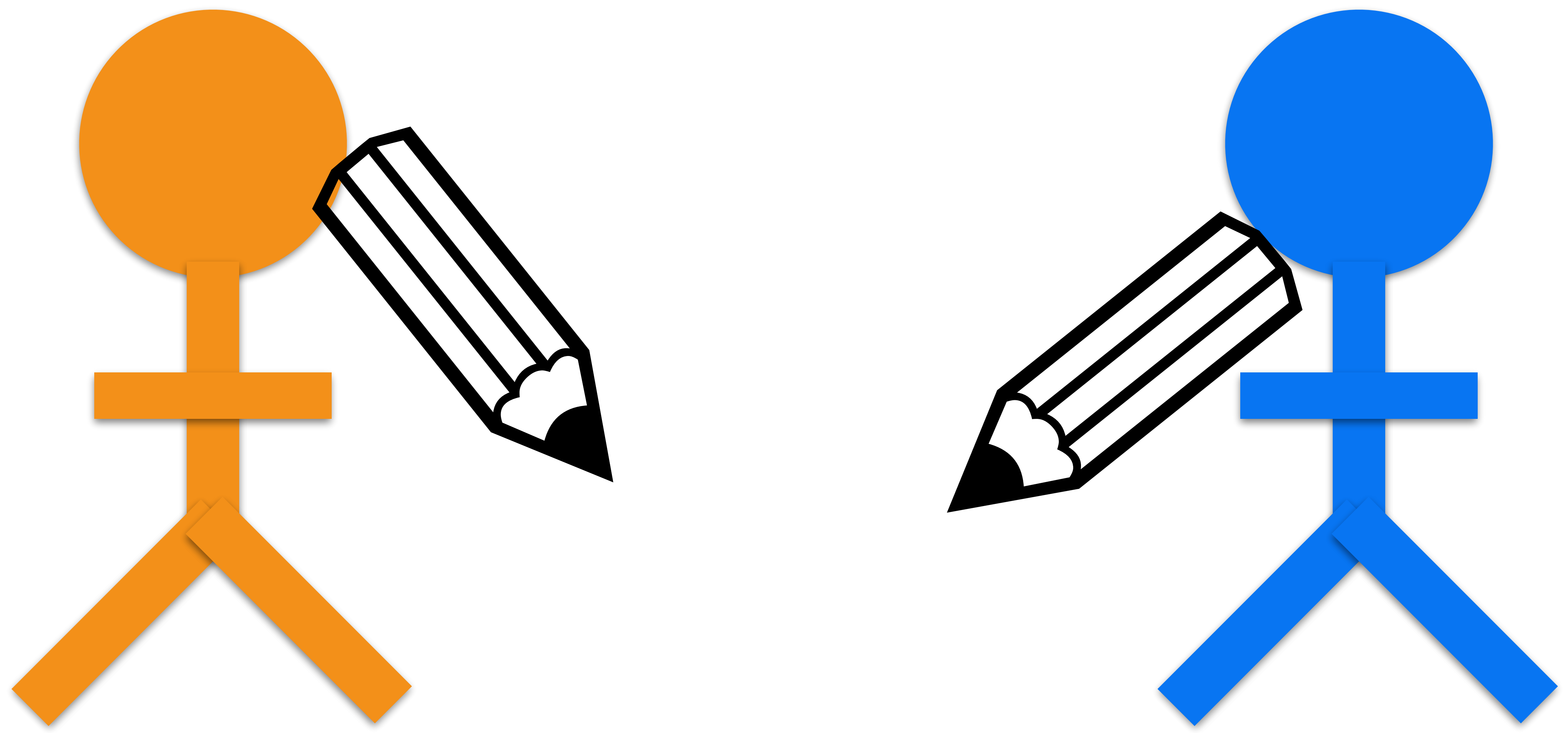
mutable shared memory

No data races,
No matter what.

Data race?

Two **unsynchronized** threads
accessing **same data**
where **at least one writes**.

Data race?



Two **unsynchronized** threads
accessing **same data**
where **at least one writes**.

Aliasing

+ Mutation

+ No ordering

Data race

Sound familiar?

Aliasing

+ Mutation

+ No ordering

Data race

No data races =
No accidentally-shared state.

All sharing is explicit!

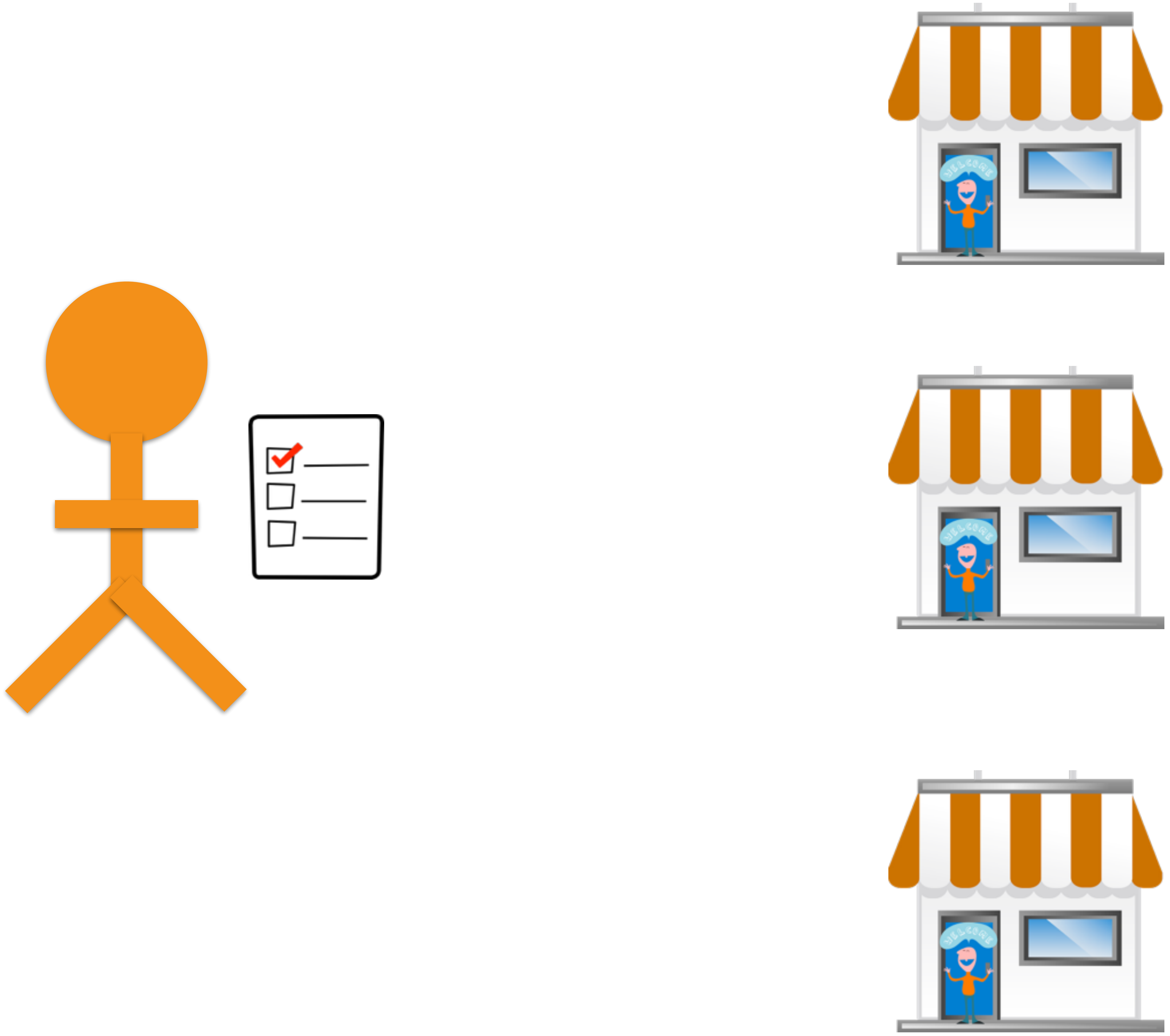
No data races =
No accidentally-shared state.

All sharing is explicit!

```
// Assume some_value: &mut i32

*some_value = 5;
return *some_value == 5;  // ALWAYS true
```

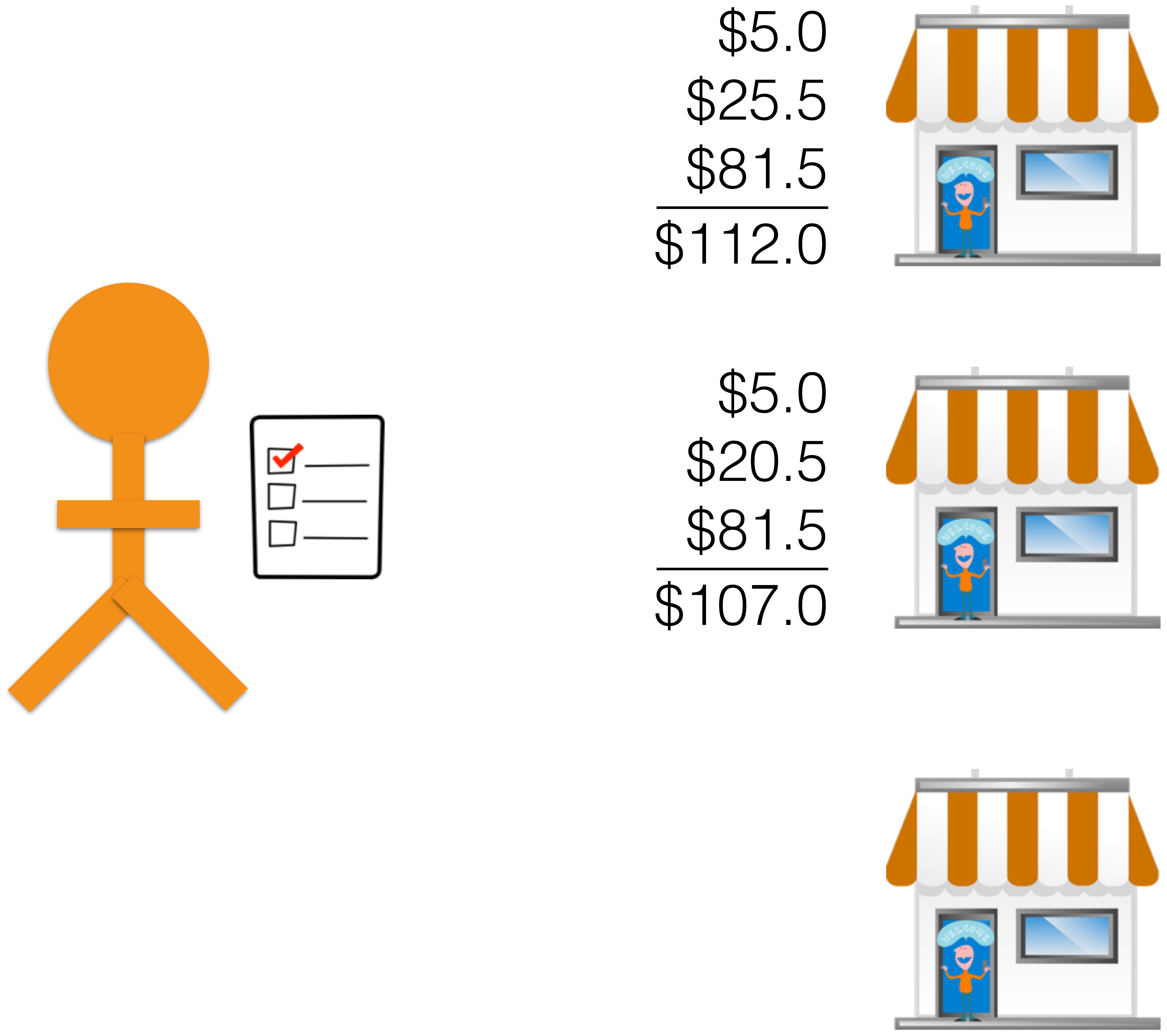
By QuentinUK (Own work), via Wikimedia Commons

Shop till you drop!



$5.0
$25.5
$81.5

$112.0


$5.0
$25.5
$81.5
$112.0
$5.0
$20.5
$81.5
$107.0

$5.0
$25.5
$81.5
$112.0

$5.0
$20.5
$81.5
$107.0

$5.0
$23.5
XXX
XXX

$5.0
$25.5
$81.5
$112.0

$5.0
$20.5
$81.5
$107.0

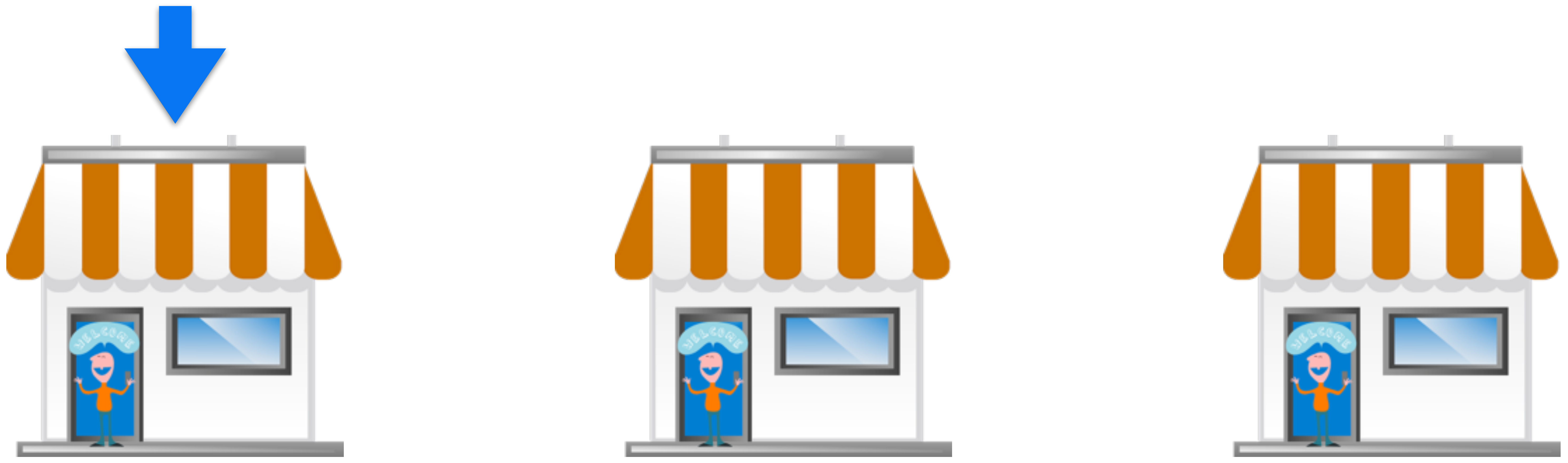
★

$5.0
$23.5
XXX
XXX

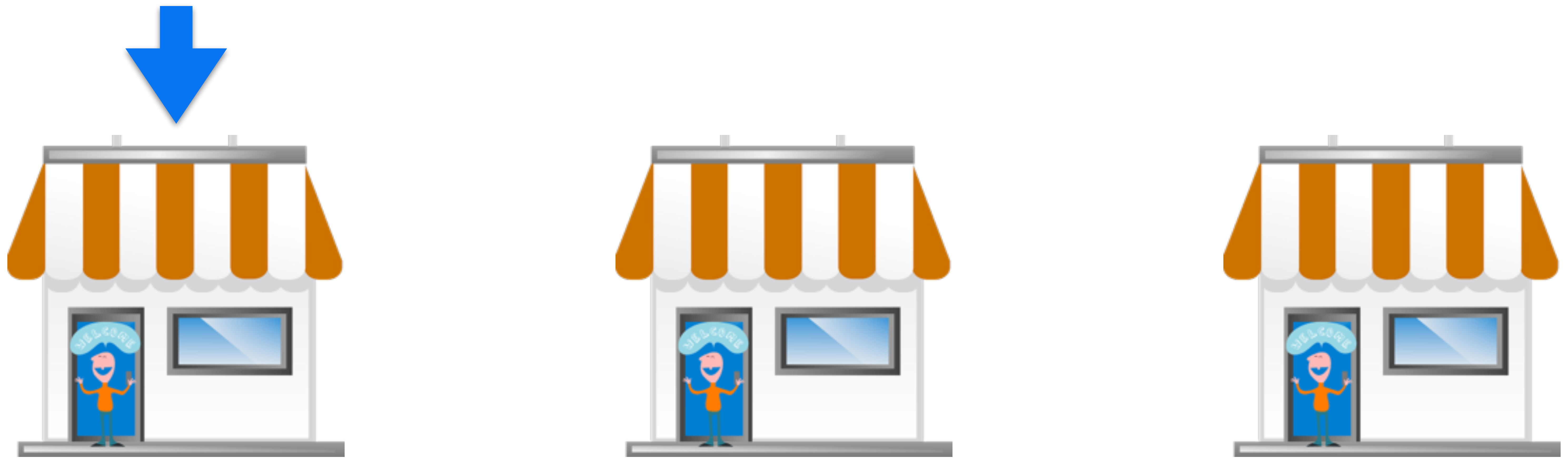
```
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```

```
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```

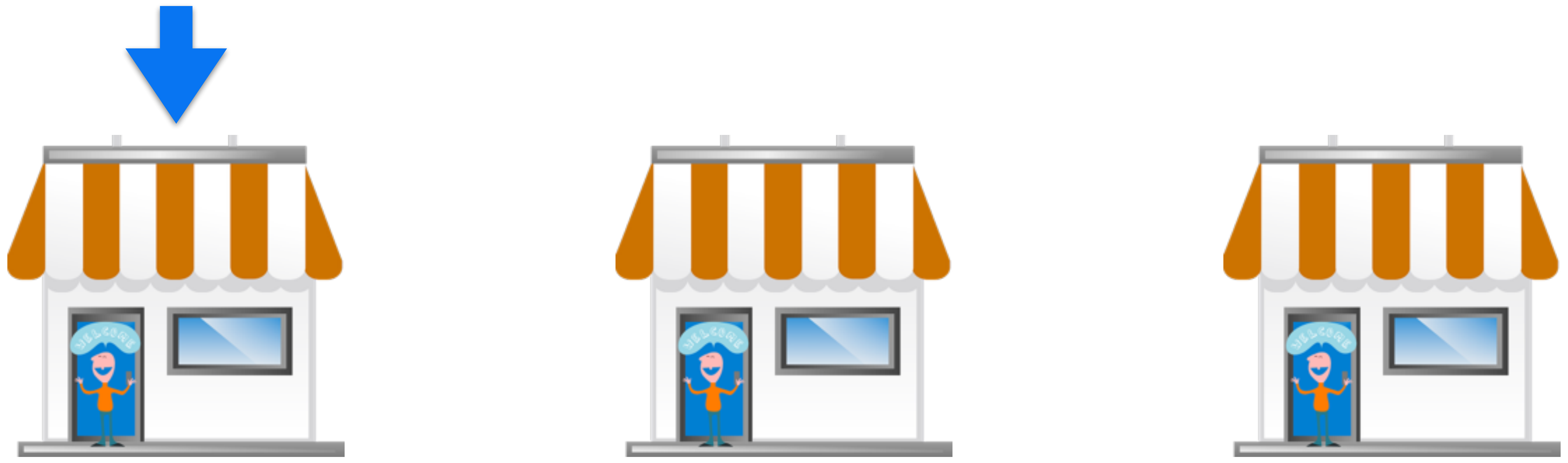
```
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```



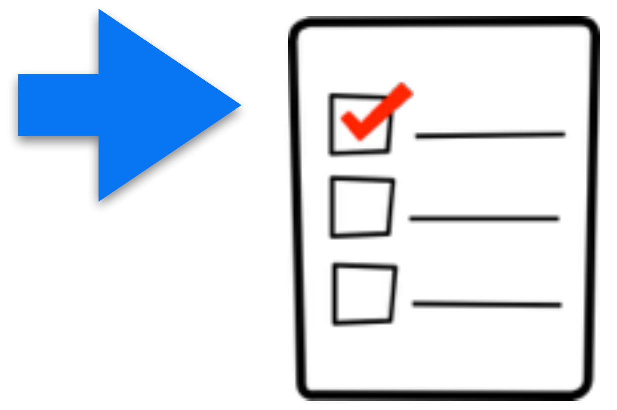
```rust
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```

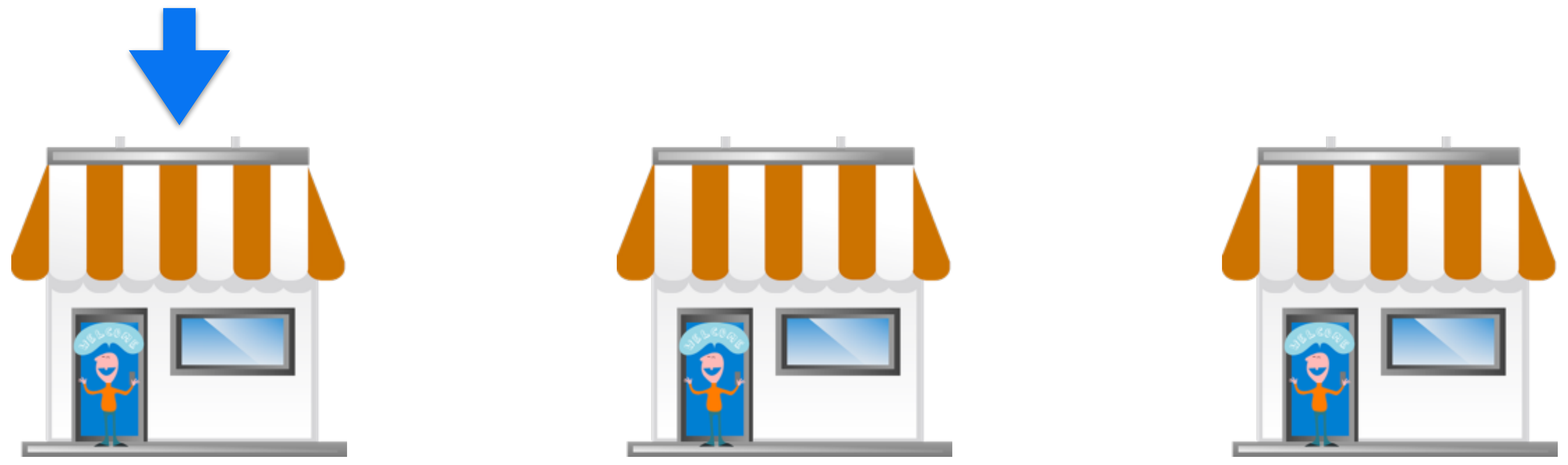


```
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```

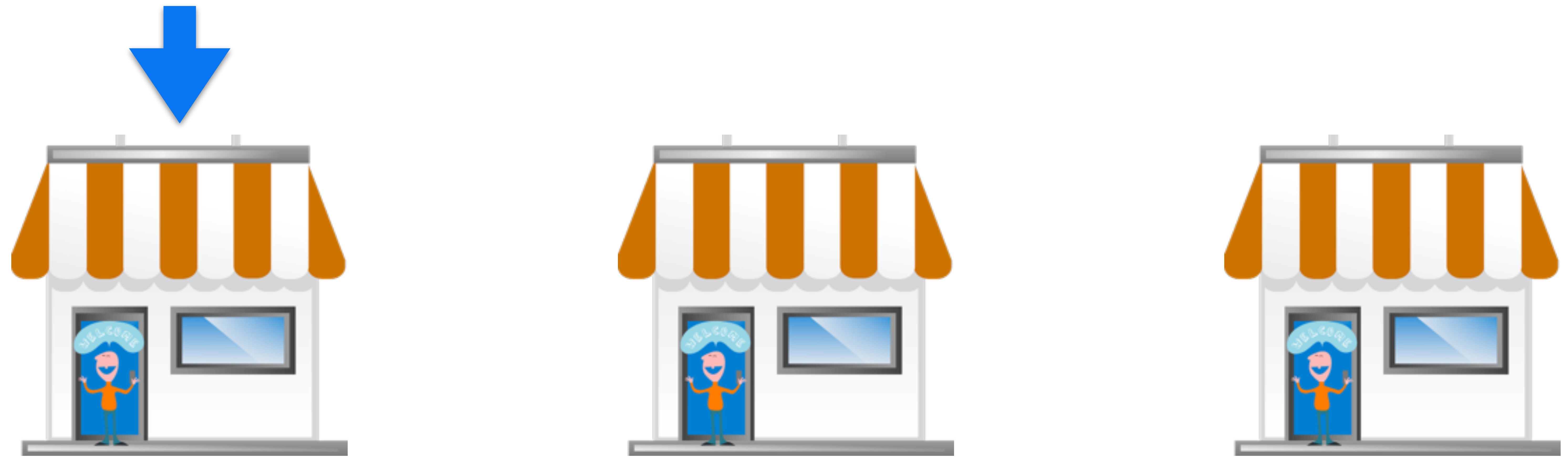


```
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```

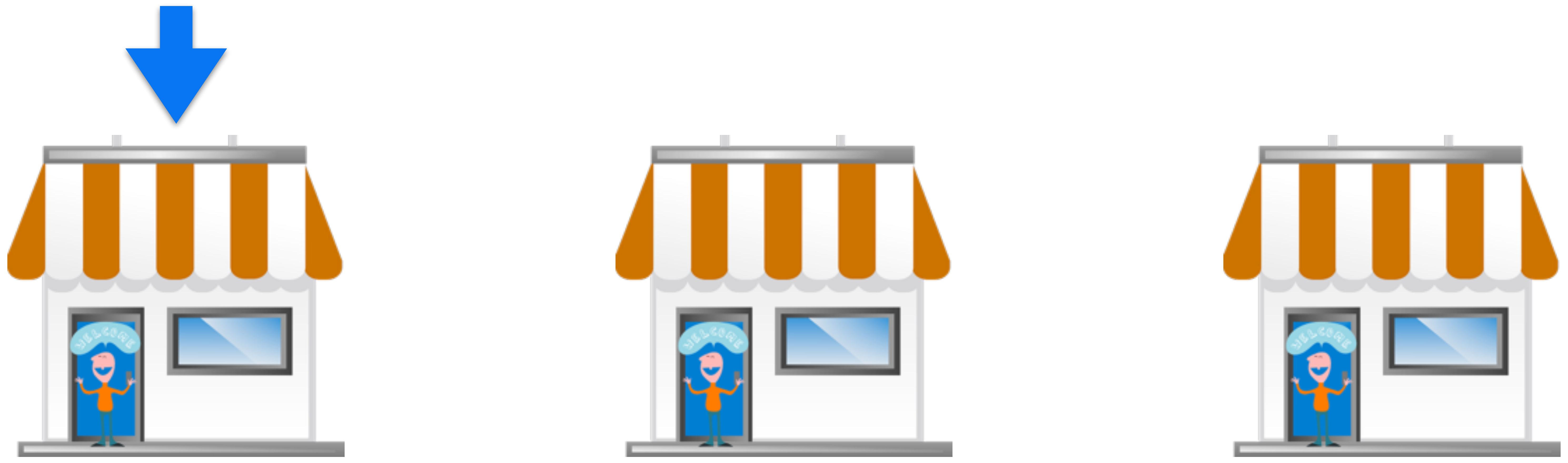




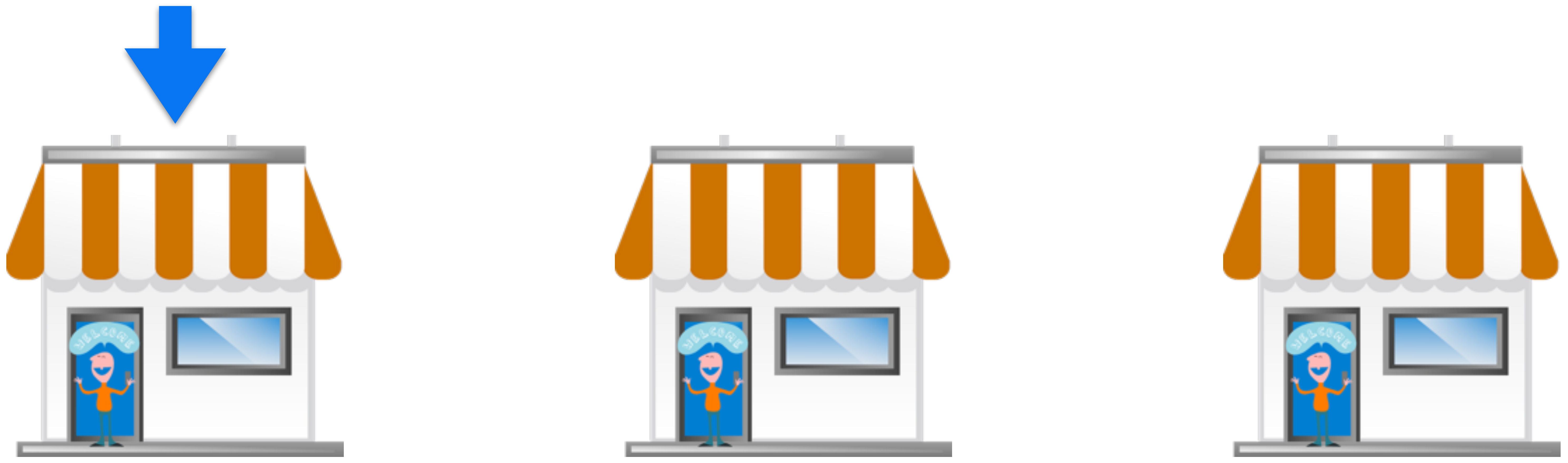
```
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```



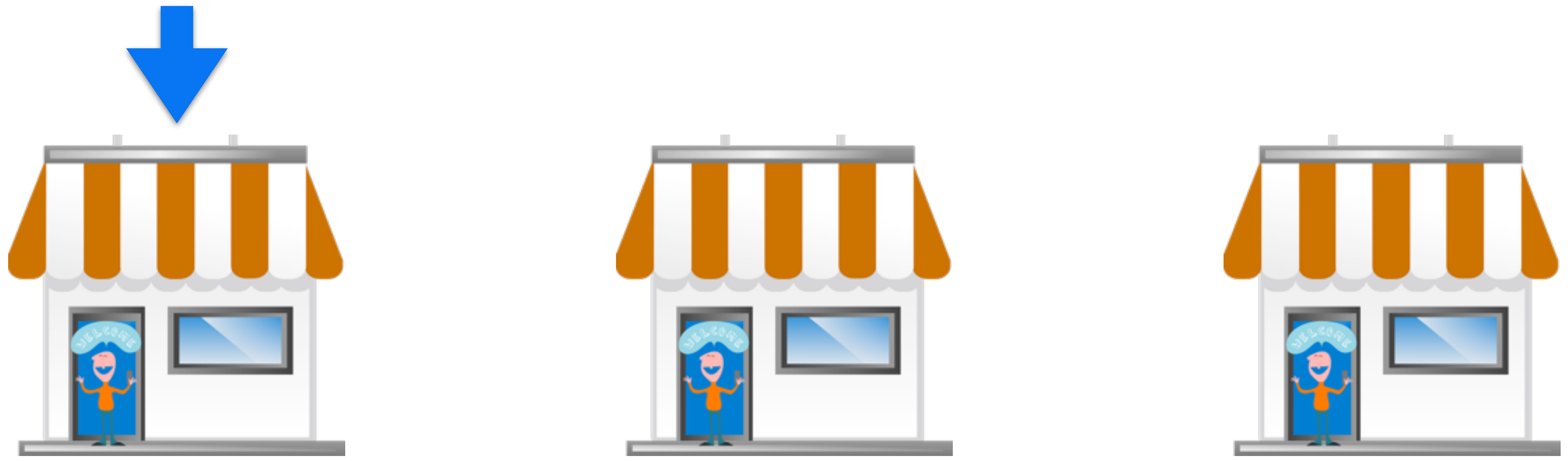
```
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```



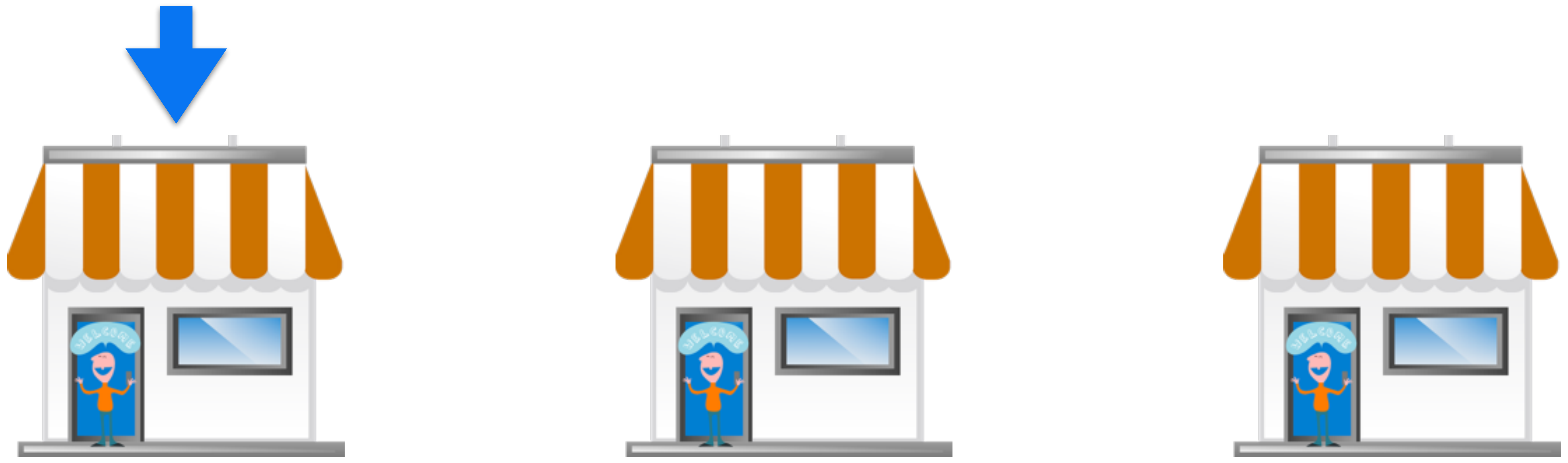
```
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```



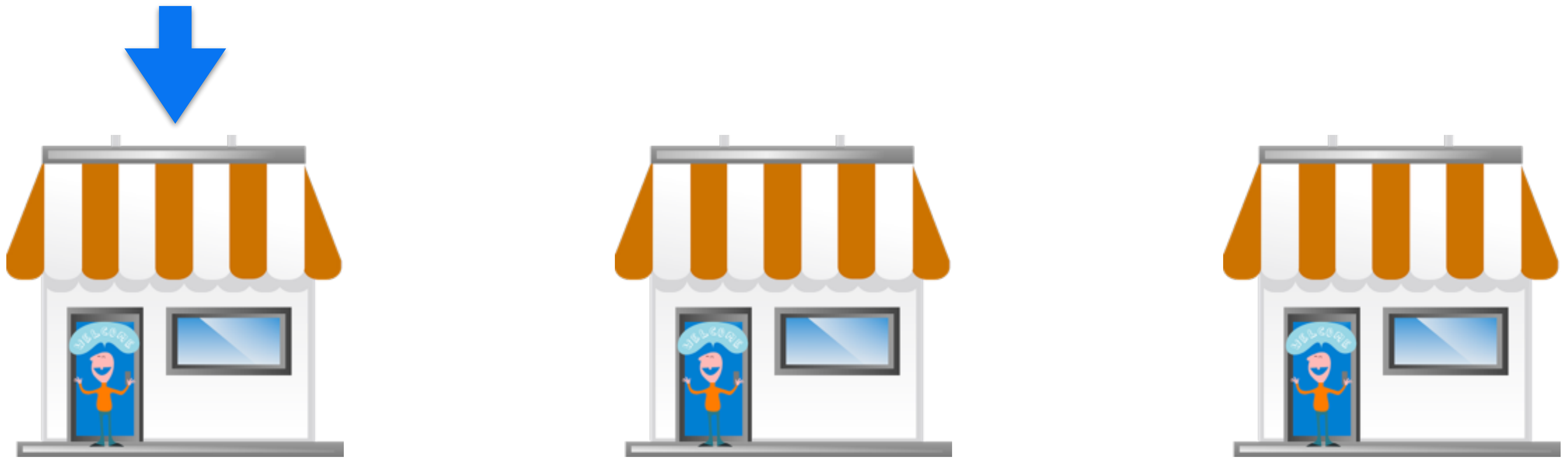
```
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```



```
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```



```
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```



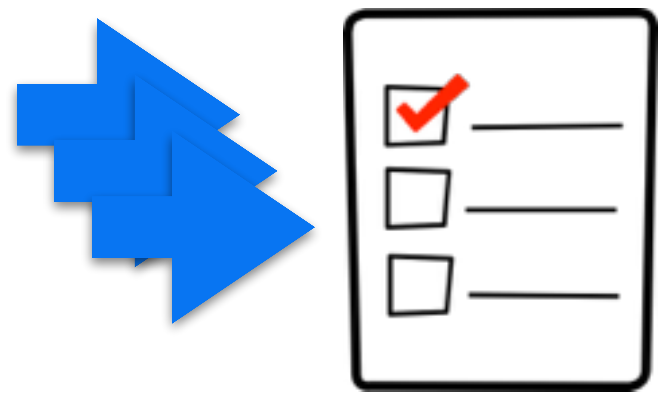
```
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```

```
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```

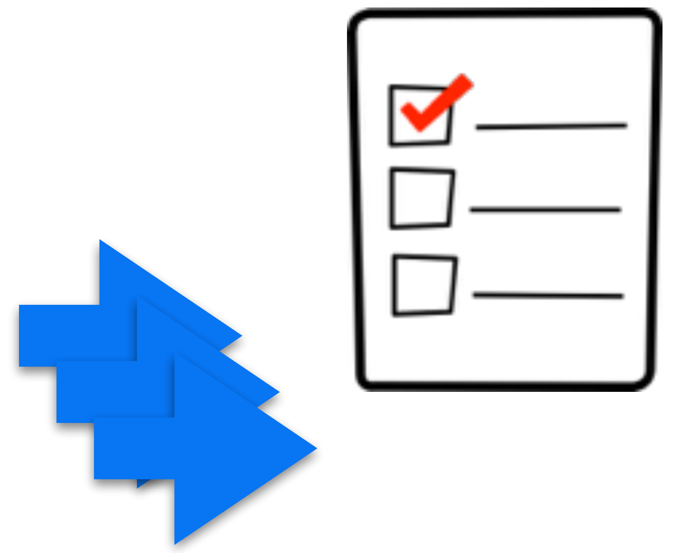
```
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```

```
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```

```
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```



```
let mut best = None;
let mut best_price = INFINITY;
for store in stores {
  let sum = store.total_price(&shopping_list);
  match sum {
    Some(s) if s < best_price => {
      best = Some(store.name);
      best_price = s;
    }
    _ => { }
  }
}
```



```
use std::thread;
…
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```rust
use std::thread;
…
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```rust
use std::thread;
…
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

Closure **takes ownership** of variables it uses.

```
use std::thread;
…
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

Closure **takes ownership** of variables it uses.

Variables used by this closure.

```
use std::thread;
…
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

Closure **takes ownership** of variables it uses.

Variables used by this closure.

```
use std::thread;
…
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

Handle to the
thread we spawned.

```
use std::thread;
…
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

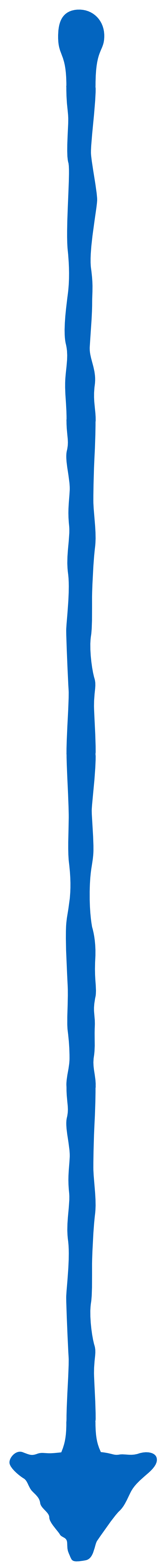
Closure body **can** produce a result: **(String, Option<f32>)**.

```
use std::thread;
…
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

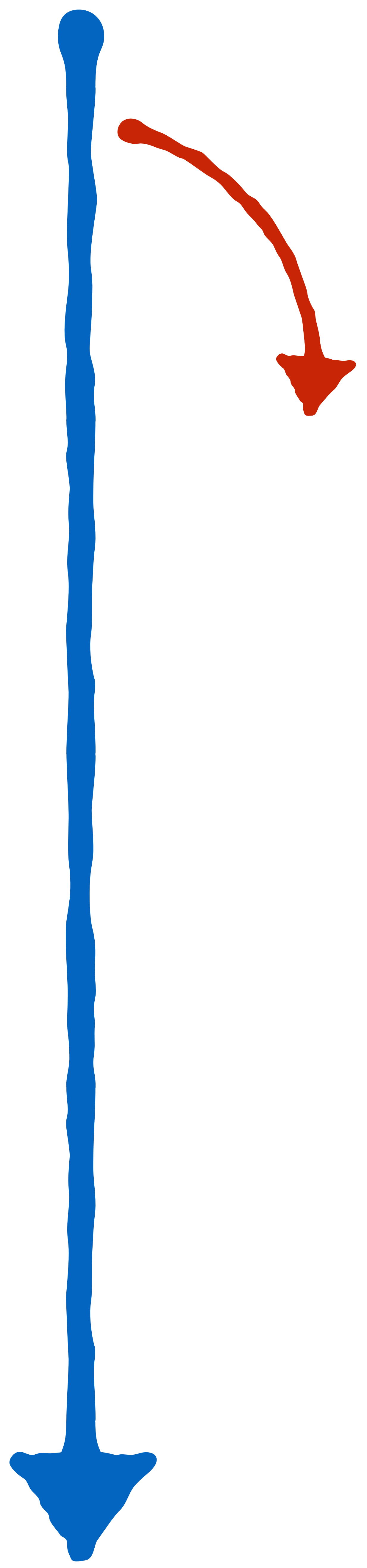
```
use std::thread;
…
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```rust
use std::thread;
…
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

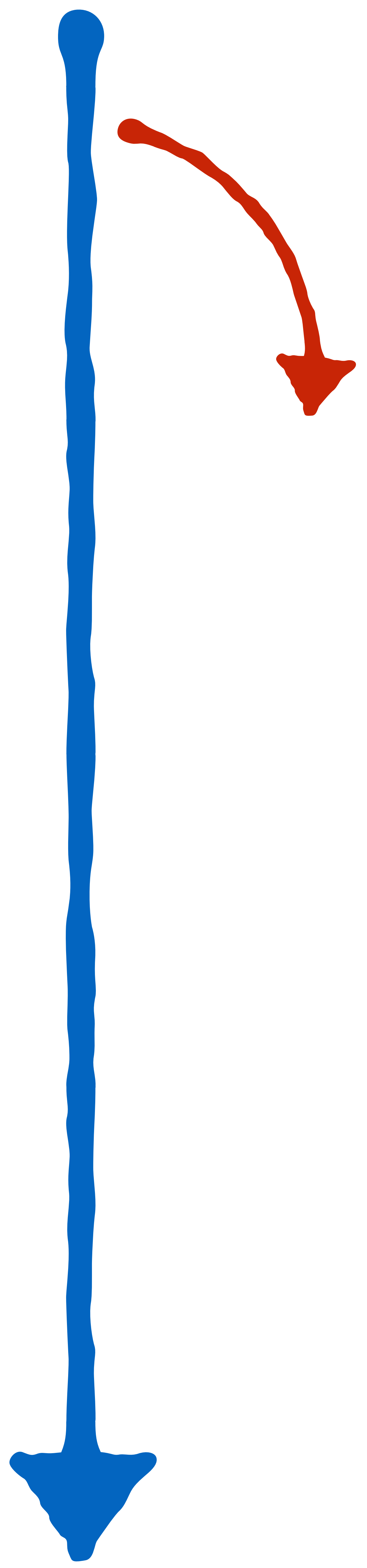
```
let handle = thread::spawn(…);
```



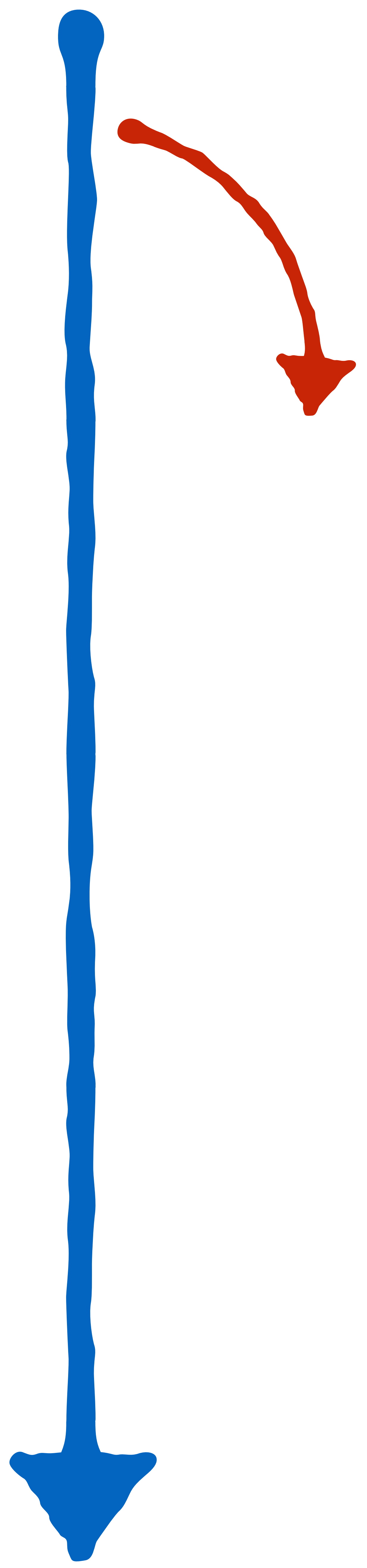
```
let handle = thread::spawn(…);
```



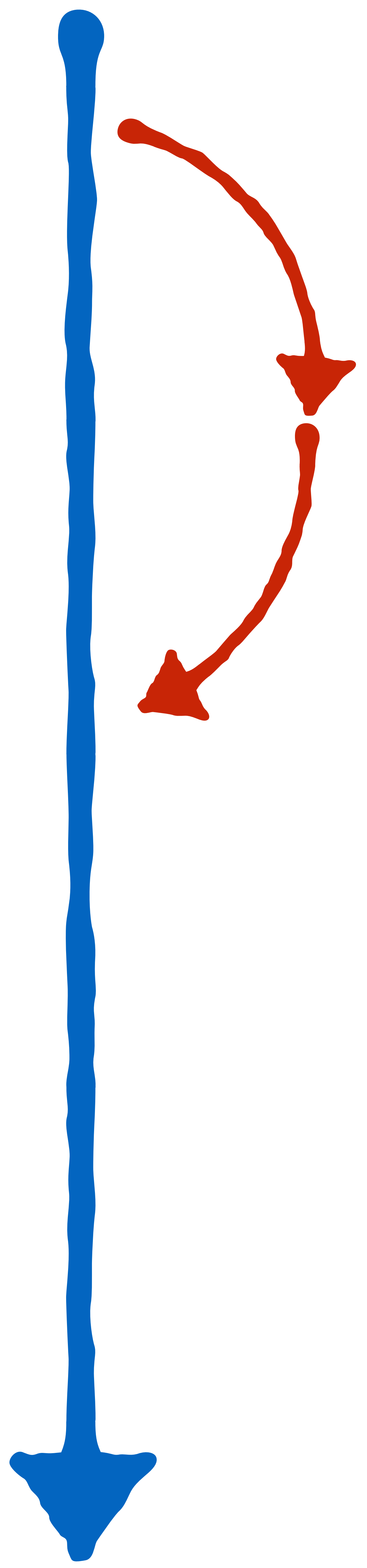
```
let handle = thread::spawn(…);

… // stuff in parallel
  // with new thread
```



```
let handle = thread::spawn(…);

… // stuff in parallel
  // with new thread

let (name, sum) = handle.join().unwrap();
```

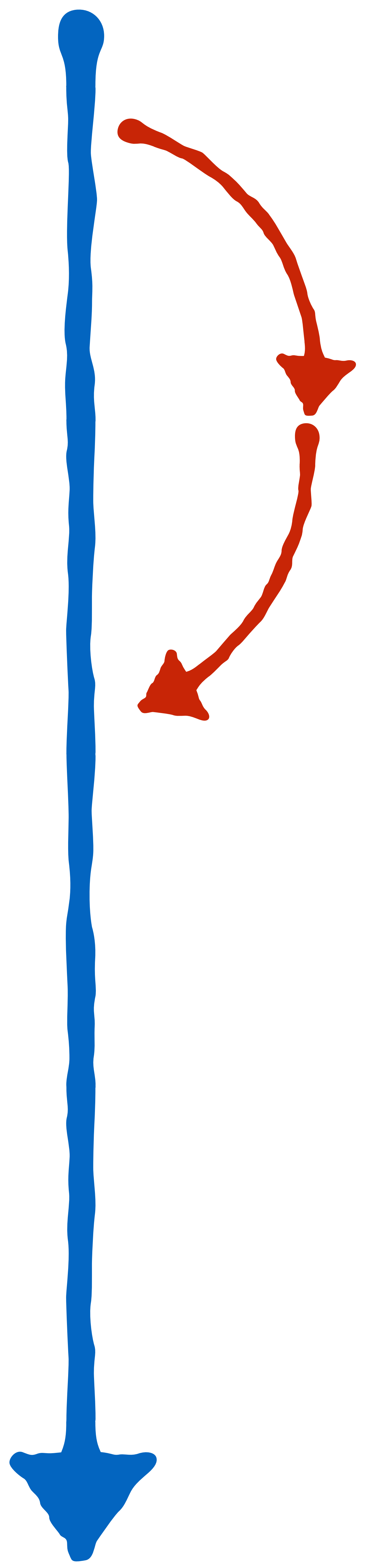


```
let handle = thread::spawn(…);

… // stuff in parallel
  // with new thread

let (name, sum) = handle.join().unwrap();
```



```
let handle = thread::spawn(…);

… // stuff in parallel
  // with new thread

let (name, sum) = handle.join().unwrap();
```

Wait for thread
to finish and
get return value.

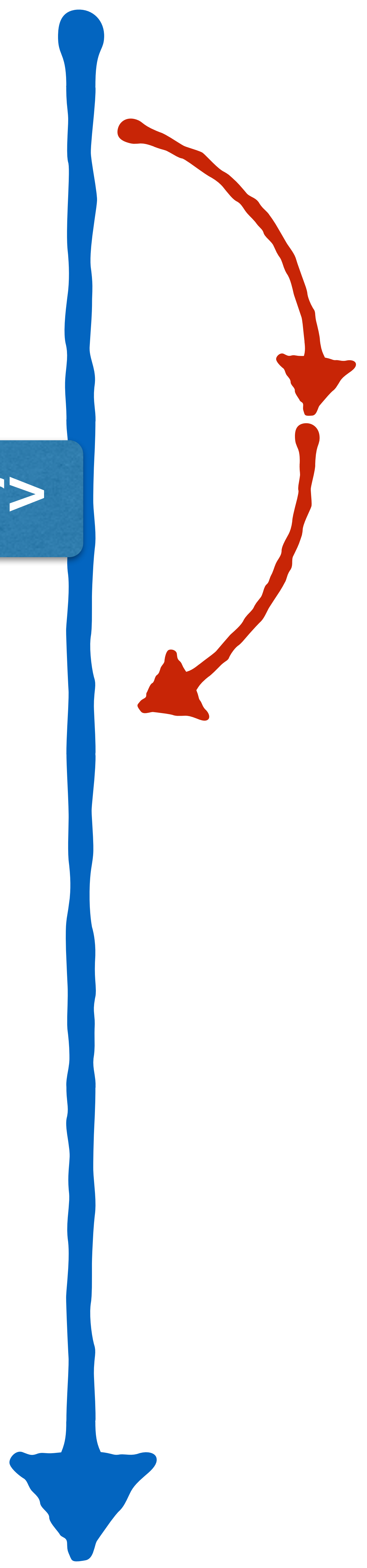


```
let handle = thread::spawn(…);

… // stuff in parallel
  // wi
```

```
Result<(String, Option<f32>), Error>
```

```
let (name, sum) = handle.join().unwrap();
```

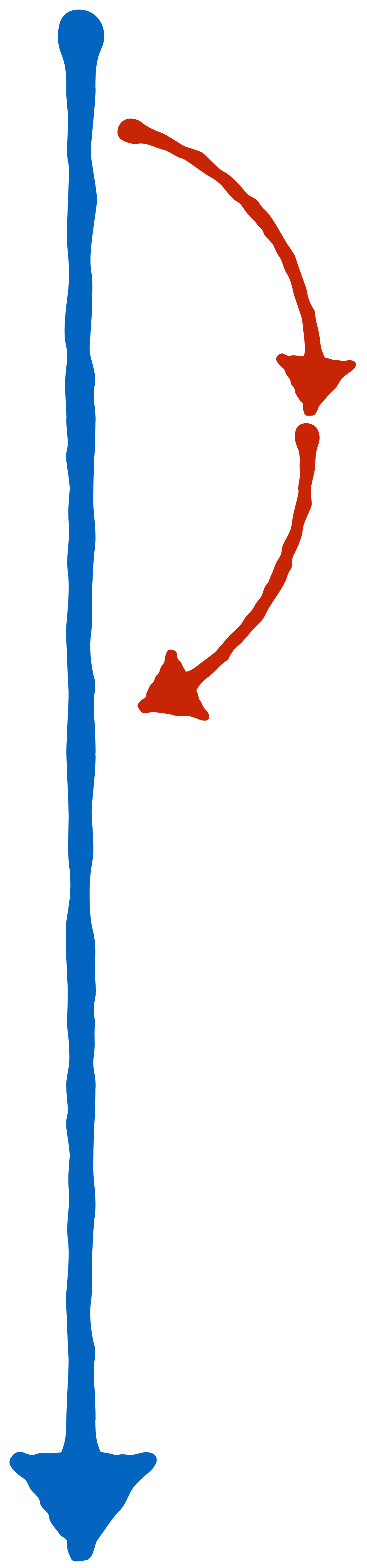
Wait for thread
to finish and
get return value.



```
let handle = thread::spawn(…);

… // stuff in parallel
  // with new thread

let (name, sum) = handle.join().unwrap();
```

Wait for thread to finish and get return value.

Thread may have panicked. Propagate.

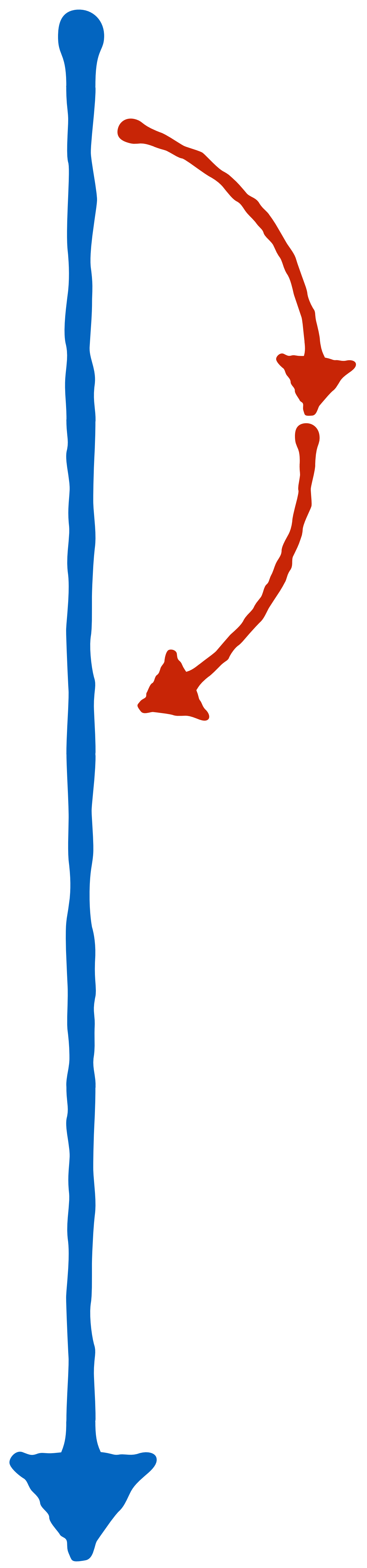


```
let handle = thread::spawn(…);

… // stuff in parallel
  // with new thread

let (name, sum) = handle.join().unwrap();
```

Wait for thread to finish and get return value.

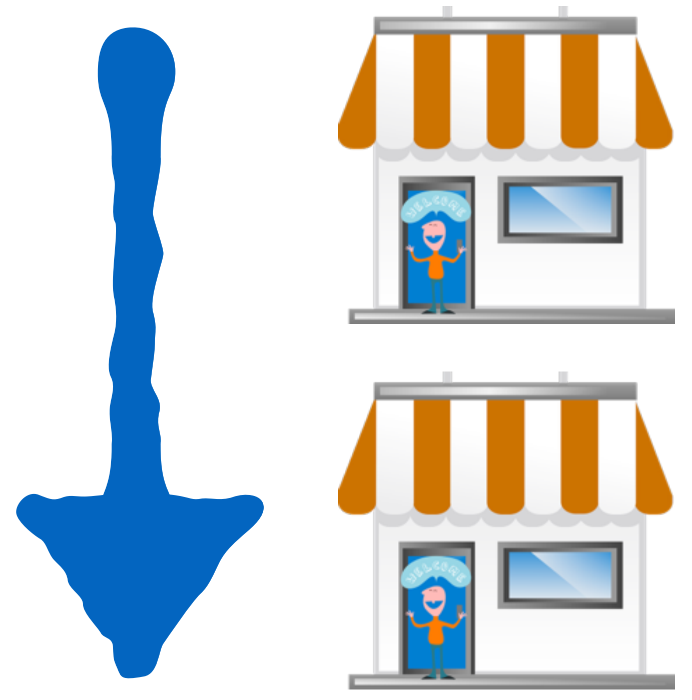
Thread may have panicked. Propagate.

Result of thread.

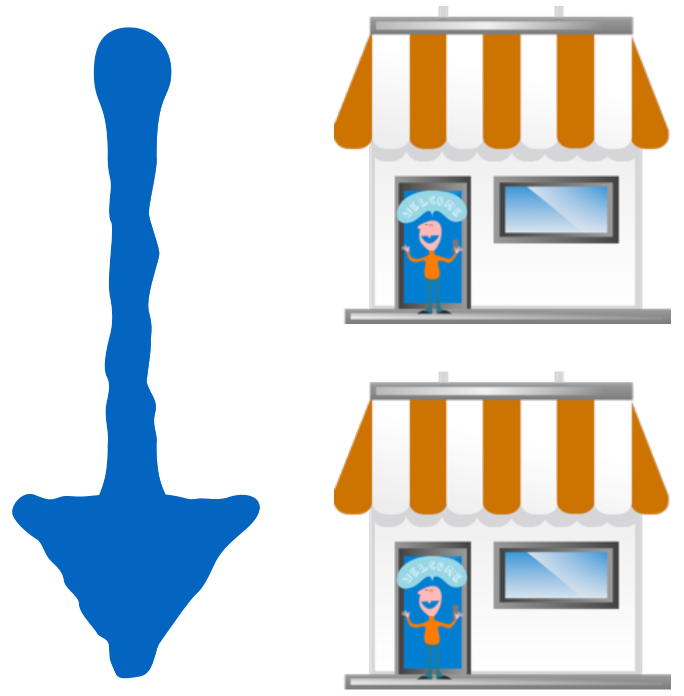


```
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  handles.push(
    thread::spawn(move || {
      let sum = …;
      (store.name, sum)
    });
}

for handle in handles {
  let (name, sum) =
    handle.join().unwrap();
  // find best price
}
```

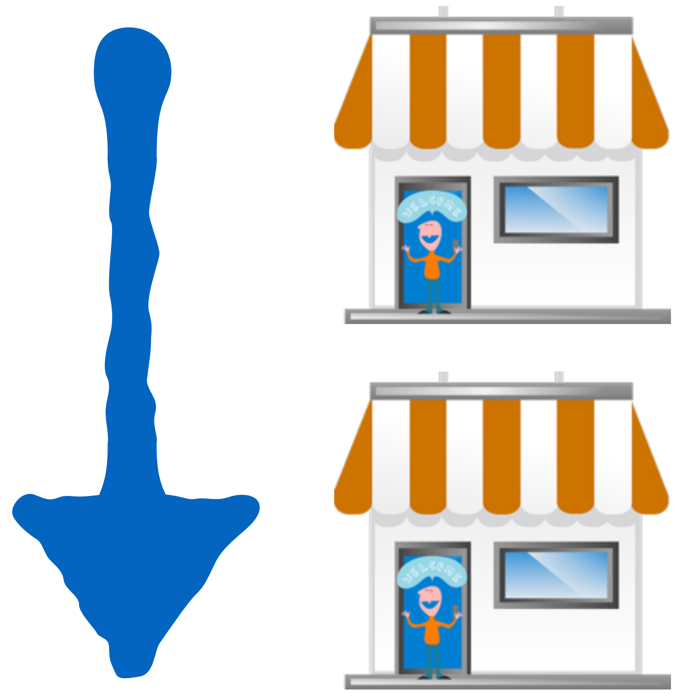
```
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  handles.push(
    thread::spawn(move || {
      let sum = …;
      (store.name, sum)
    });
}

for handle in handles {
  let (name, sum) =
    handle.join().unwrap();
  // find best price
}
```



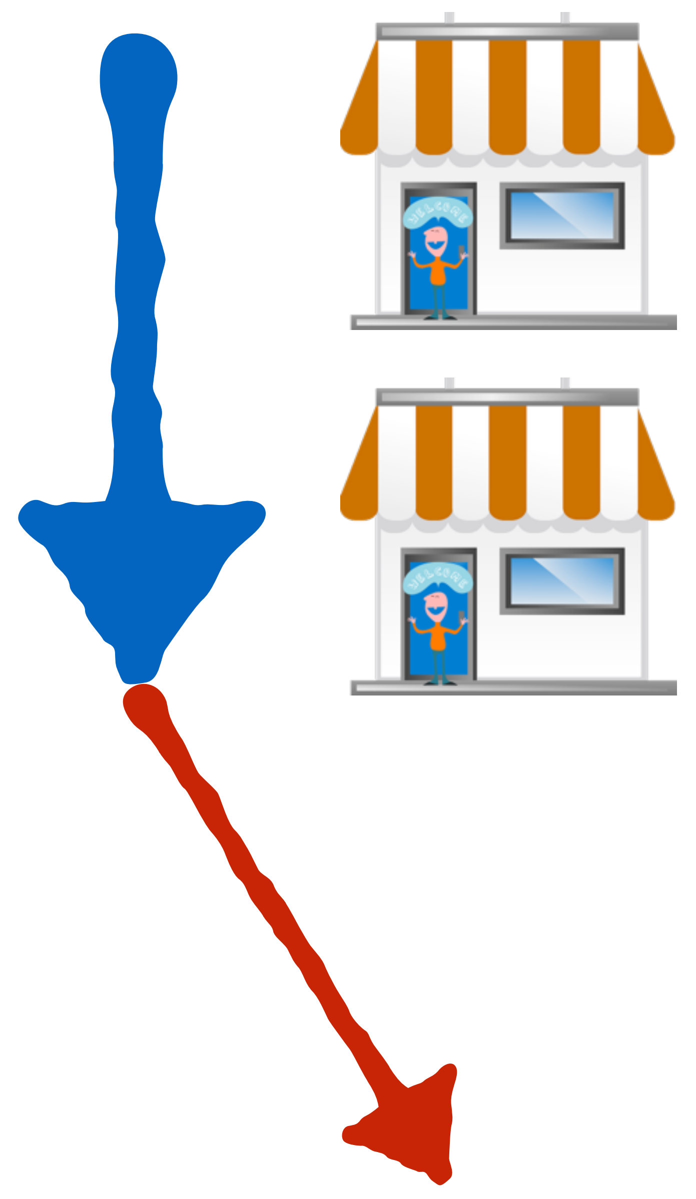
```
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  handles.push(
    thread::spawn(move || {
      let sum = …;
      (store.name, sum)
    });
}

for handle in handles {
  let (name, sum) =
    handle.join().unwrap();
  // find best price
}
```



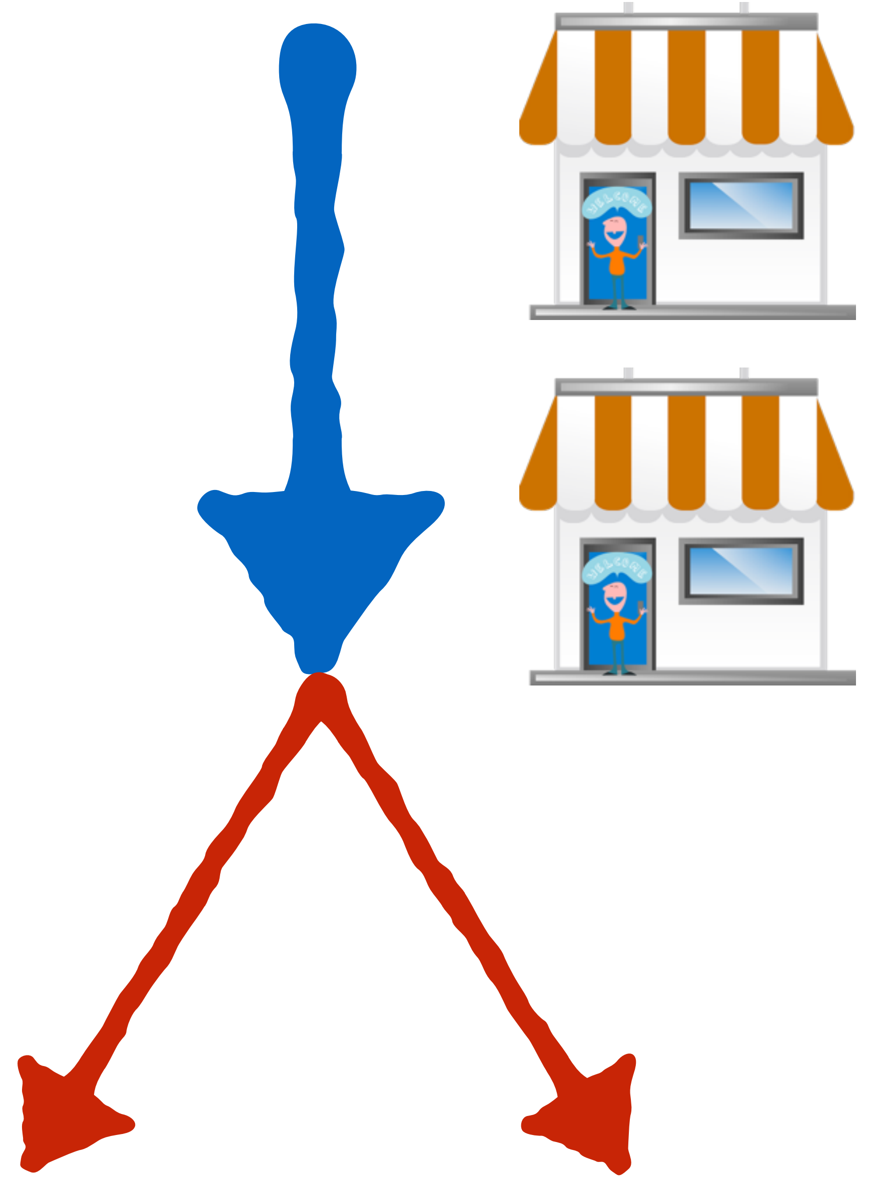
```
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  handles.push(
    thread::spawn(move || {
      let sum = …;
      (store.name, sum)
    });
}

for handle in handles {
  let (name, sum) =
    handle.join().unwrap();
  // find best price
}
```



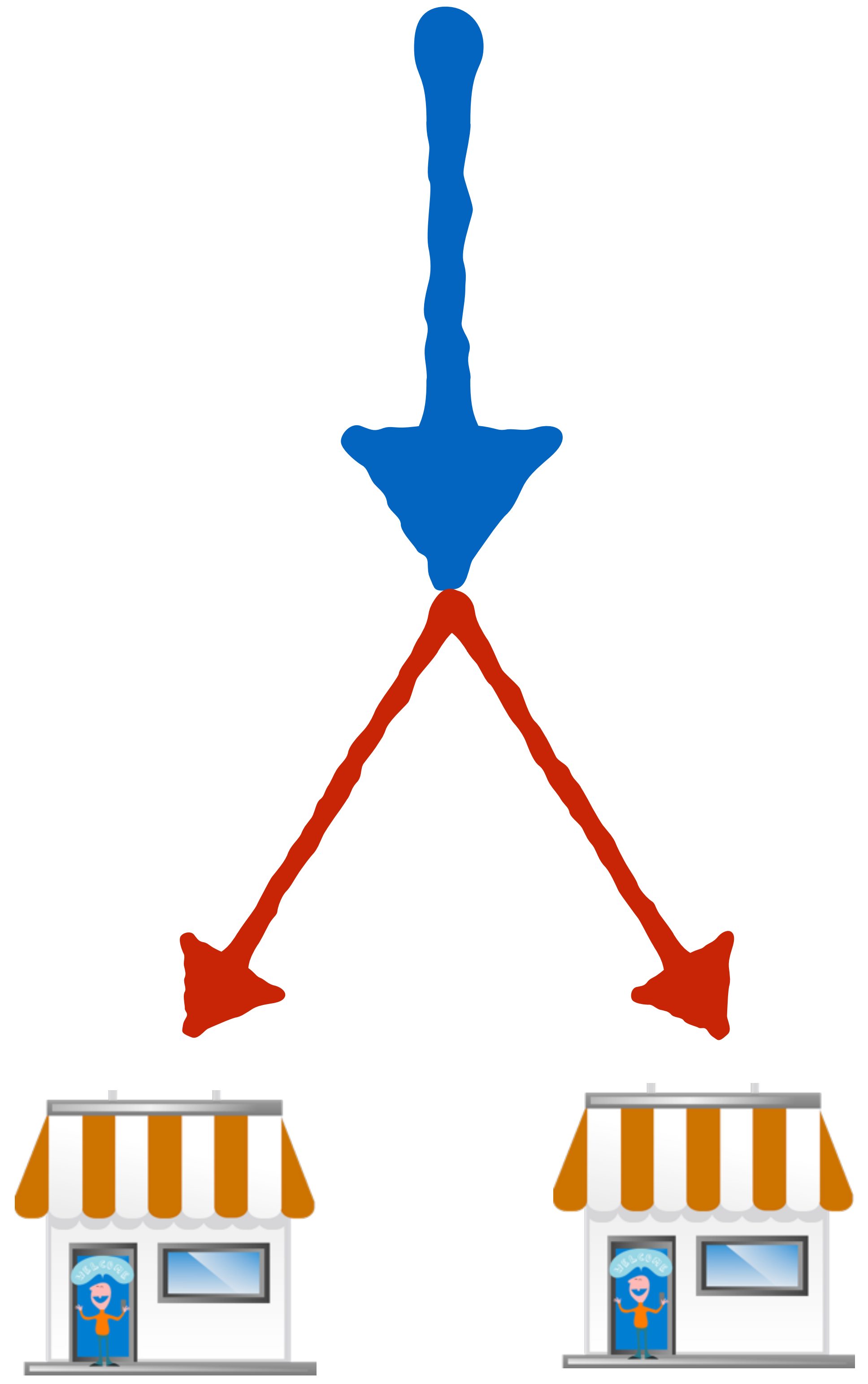
```
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  handles.push(
    thread::spawn(move || {
      let sum = …;
      (store.name, sum)
    });
}

for handle in handles {
  let (name, sum) =
    handle.join().unwrap();
  // find best price
}
```



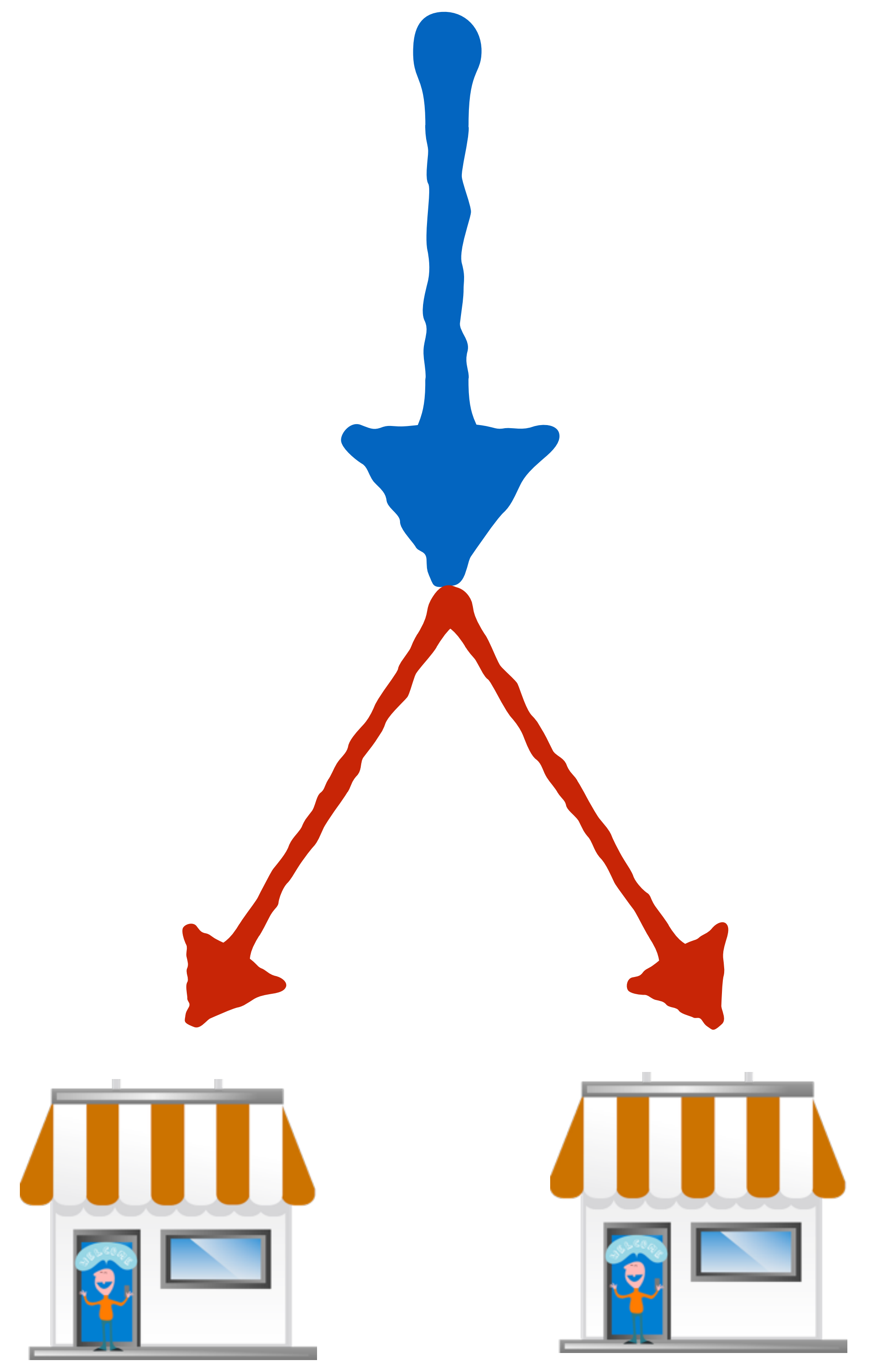
```
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  handles.push(
    thread::spawn(move || {
      let sum = …;
      (store.name, sum)
    });
}

for handle in handles {
  let (name, sum) =
    handle.join().unwrap();
  // find best price
}
```



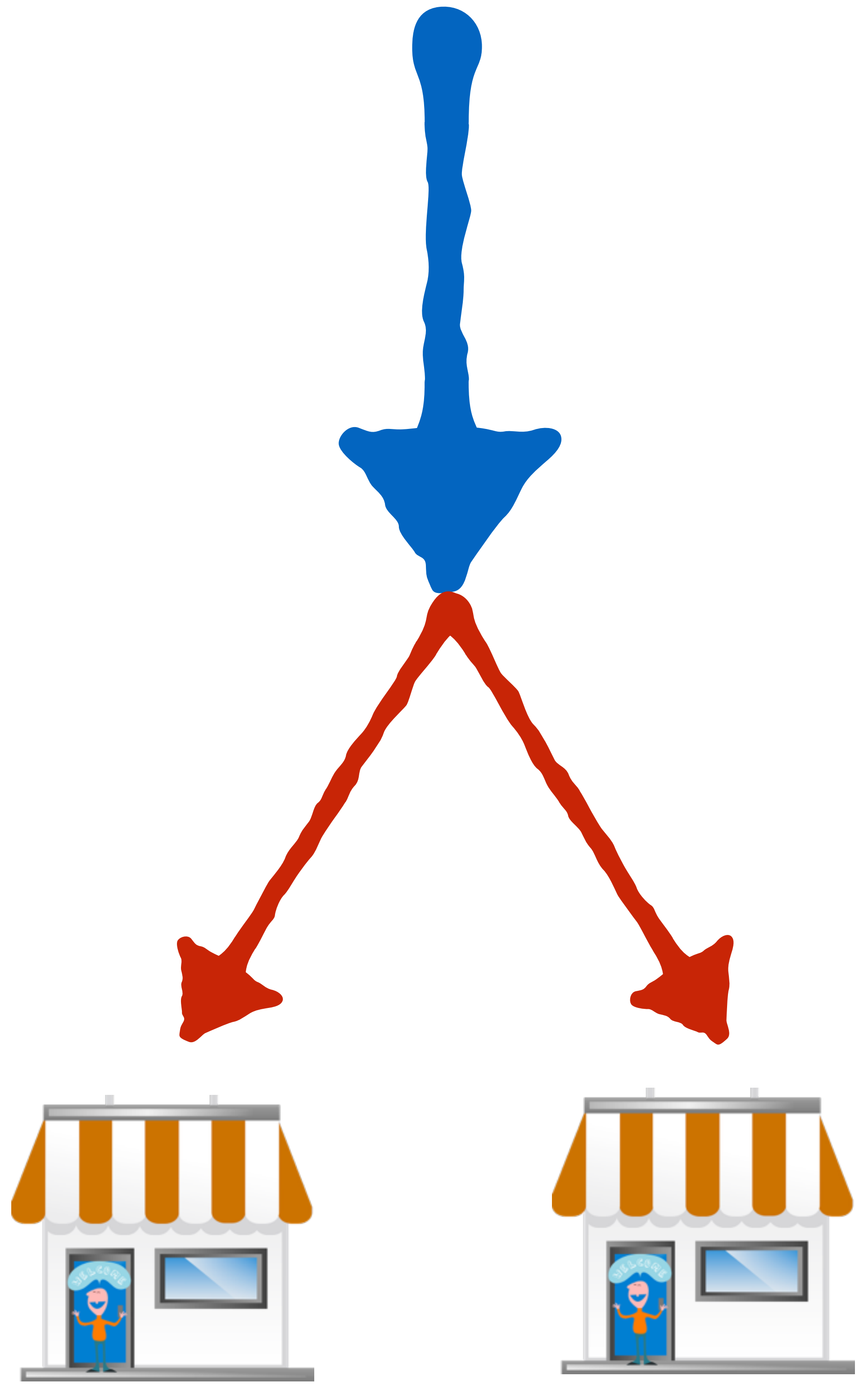
```
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  handles.push(
    thread::spawn(move || {
      let sum = …;
      (store.name, sum)
    });
}

for handle in handles {
  let (name, sum) =
    handle.join().unwrap();
  // find best price
}
```



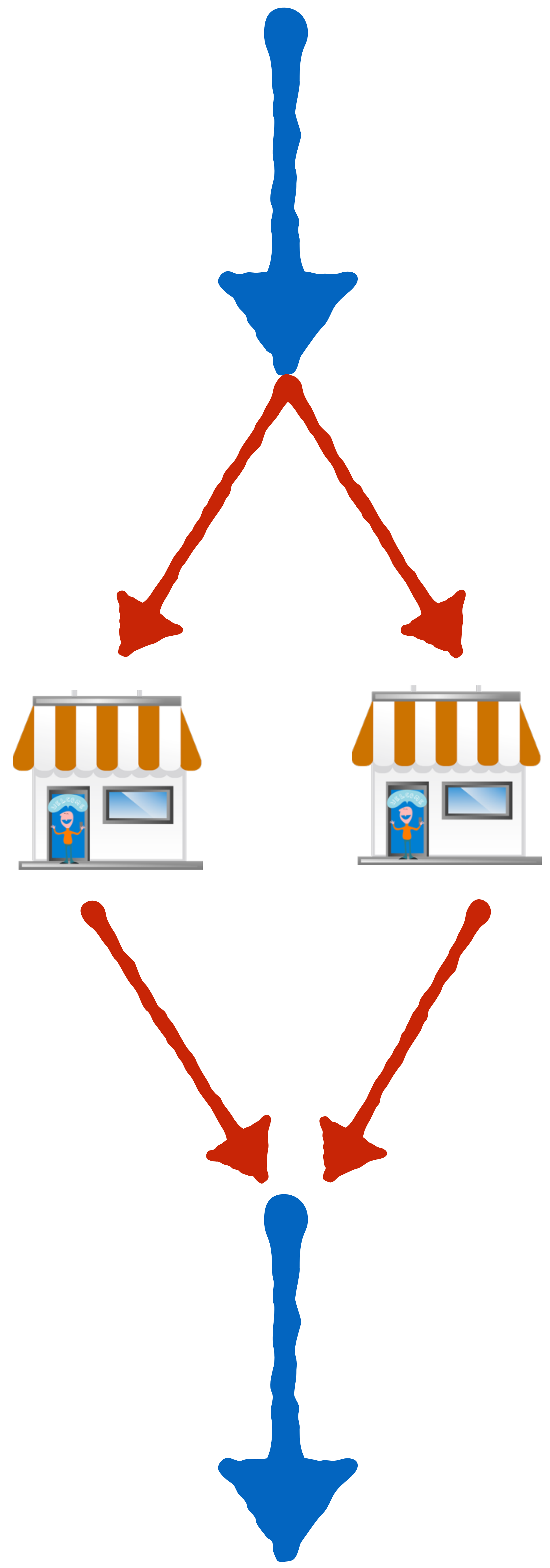
```rust
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  handles.push(
    thread::spawn(move || {
      let sum = …;
      (store.name, sum)
    });
}

for handle in handles {
  let (name, sum) =
    handle.join().unwrap();
  // find best price
}
```



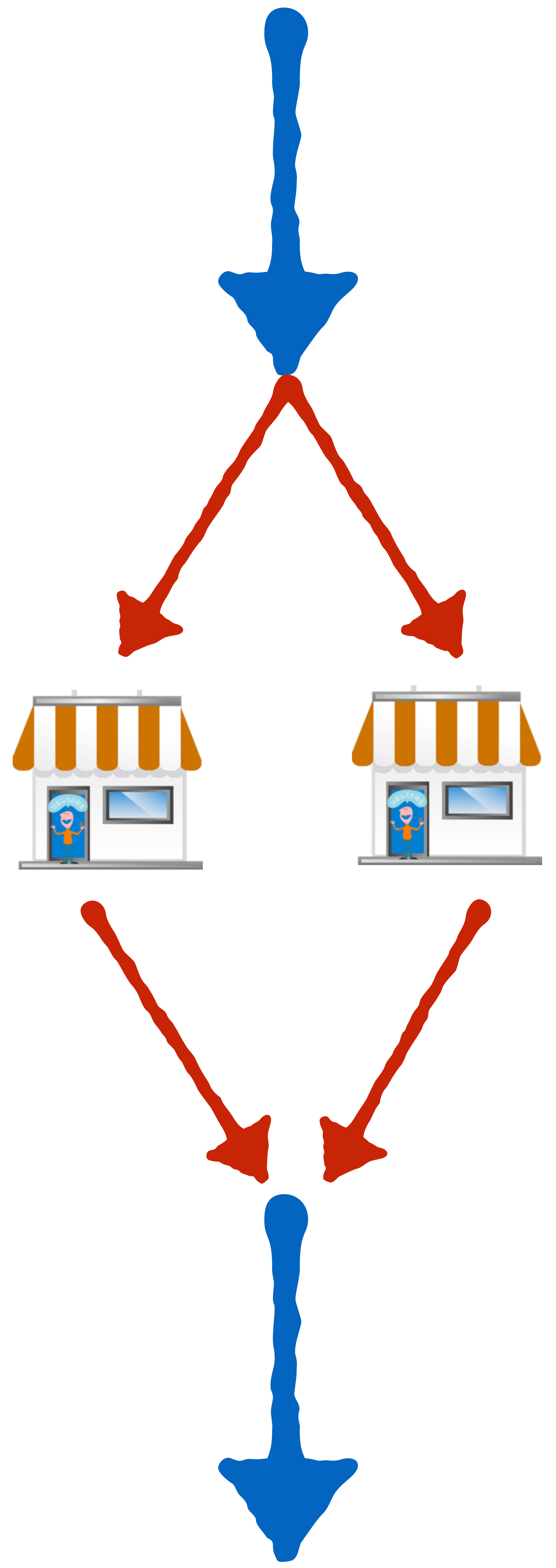
```
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  handles.push(
    thread::spawn(move || {
      let sum = …;
      (store.name, sum)
    });
}

for handle in handles {
  let (name, sum) =
    handle.join().unwrap();
  // find best price
}
```



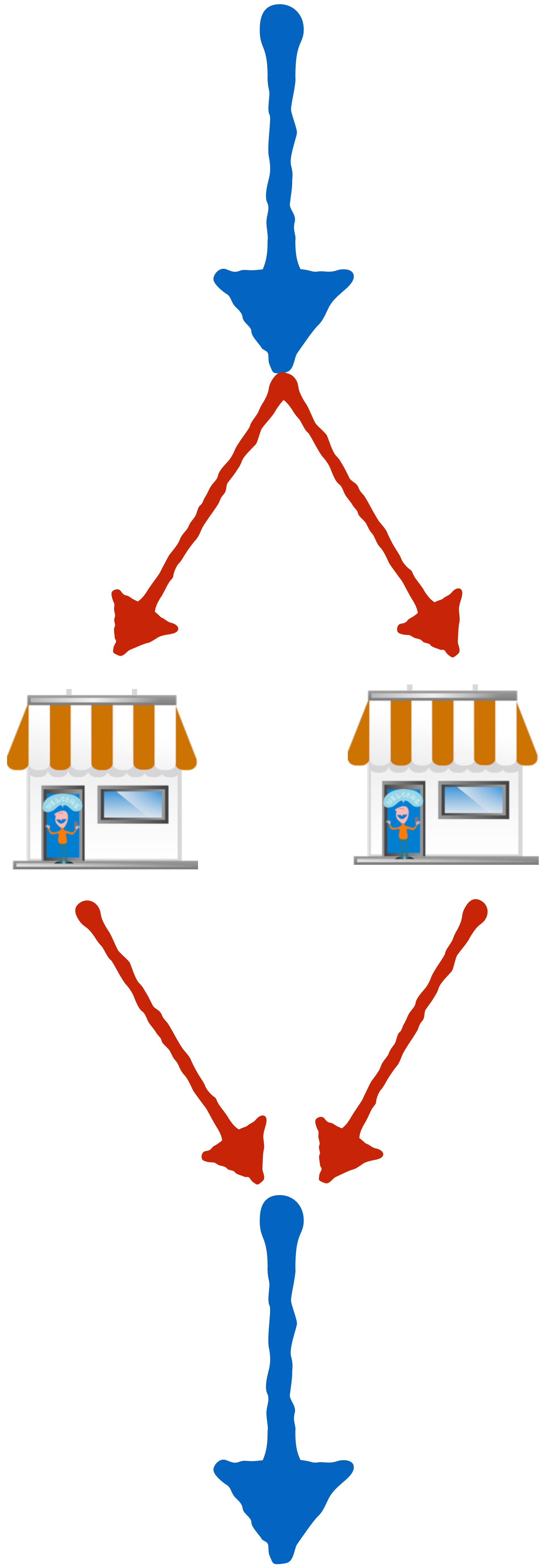
```
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  handles.push(
    thread::spawn(move || {
      let sum = …;
      (store.name, sum)
    });
}

for handle in handles {
  let (name, sum) =
    handle.join().unwrap();
  // find best price
}
```



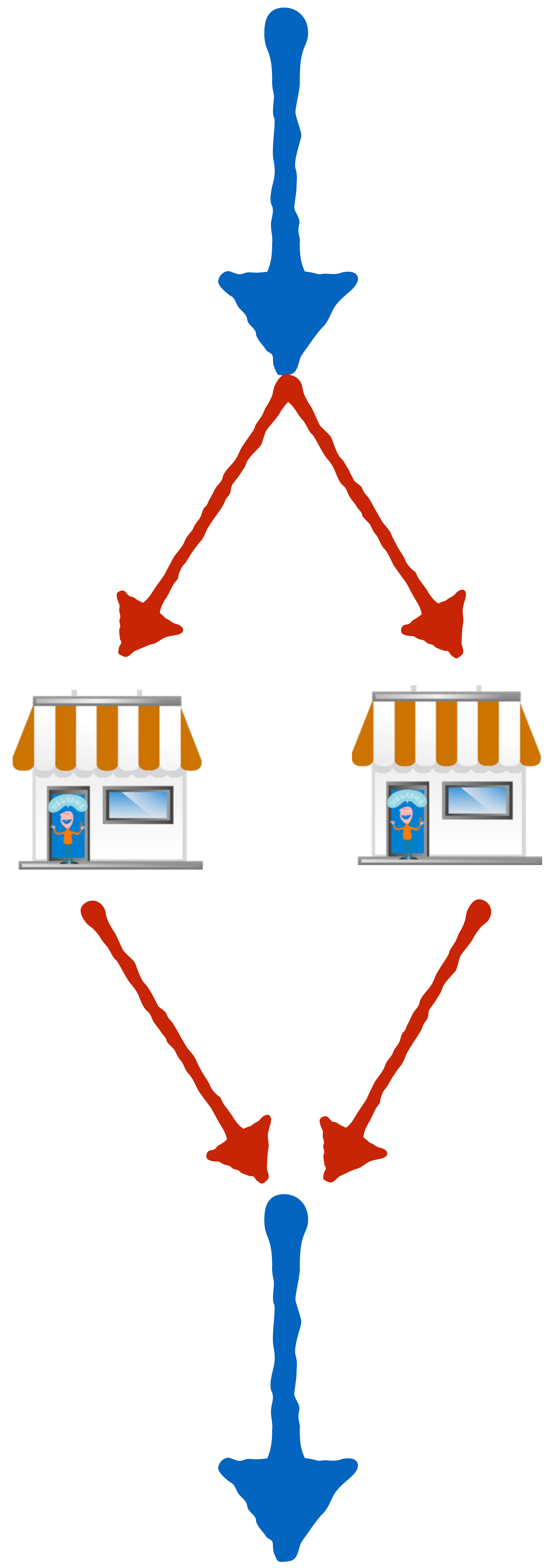
```rust
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  handles.push(
    thread::spawn(move || {
      let sum = …;
      (store.name, sum)
    });
}

for handle in handles {
  let (name, sum) =
    handle.join().unwrap();
  // find best price
}
```



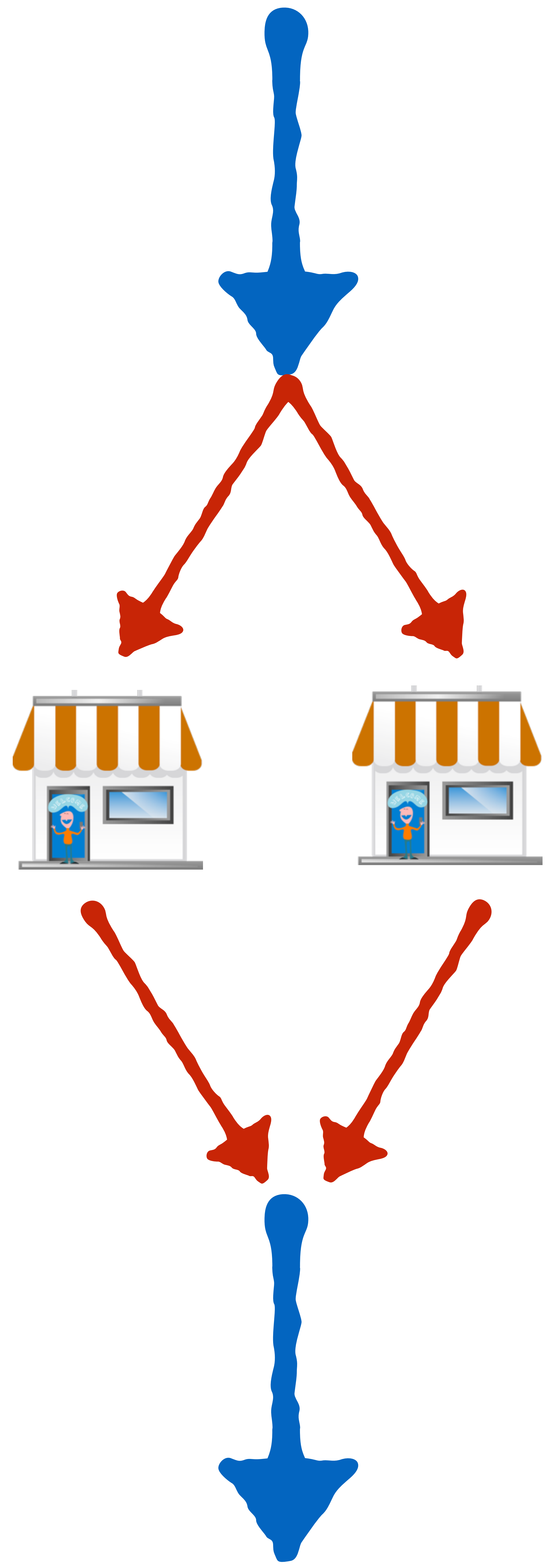
```
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  handles.push(
    thread::spawn(move || {
      let sum = …;
      (store.name, sum)
    });
}

for handle in handles {
  let (name, sum) =
    handle.join().unwrap();
  // find best price
}
```



```
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  handles.push(
    thread::spawn(move || {
      let sum = …;
      (store.name, sum)
    });
}

for handle in handles {
  let (name, sum) =
    handle.join().unwrap();
  // find best price
}
```



```
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  handles.push(
    thread::spawn(move || {
      let sum = …;
      (store.name, sum)
    });
}

for handle in handles {
  let (name, sum) =
    handle.join().unwrap();
  // find best price
}
```



```
use std::thread;
…
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

Variables used by
this closure.

```
use std::thread;
…
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
error: capture of moved value: `shopping_list`
    let sum = store.total_price(&shopping_list);
                                ^~~~~~~~~~~~~
…
help: perhaps you meant to use `clone()`?
```

```
use std::thread;
…
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
error: capture of moved value: `shopping_list`
    let sum = store.total_price(&shopping_list);
                                ^~~~~~~~~~~~~
…
help: perhaps you meant to use `clone()`?
```

```
use std::thread;
…
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```rust
use std::thread;
…
let shopping_list = vec![…];
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```rust
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```

```rust
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```

```rust
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```

```
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```

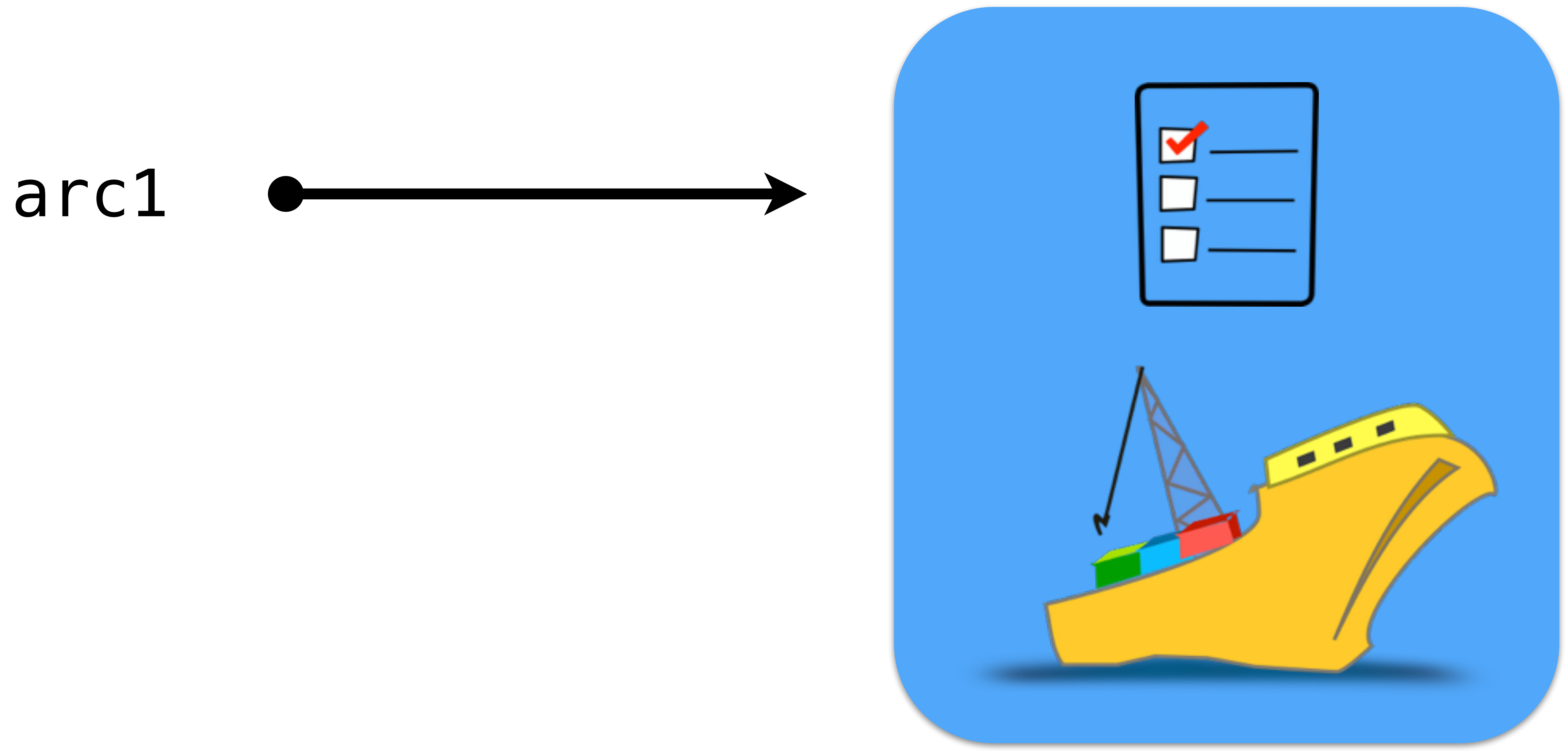
```rust
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```

```
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```

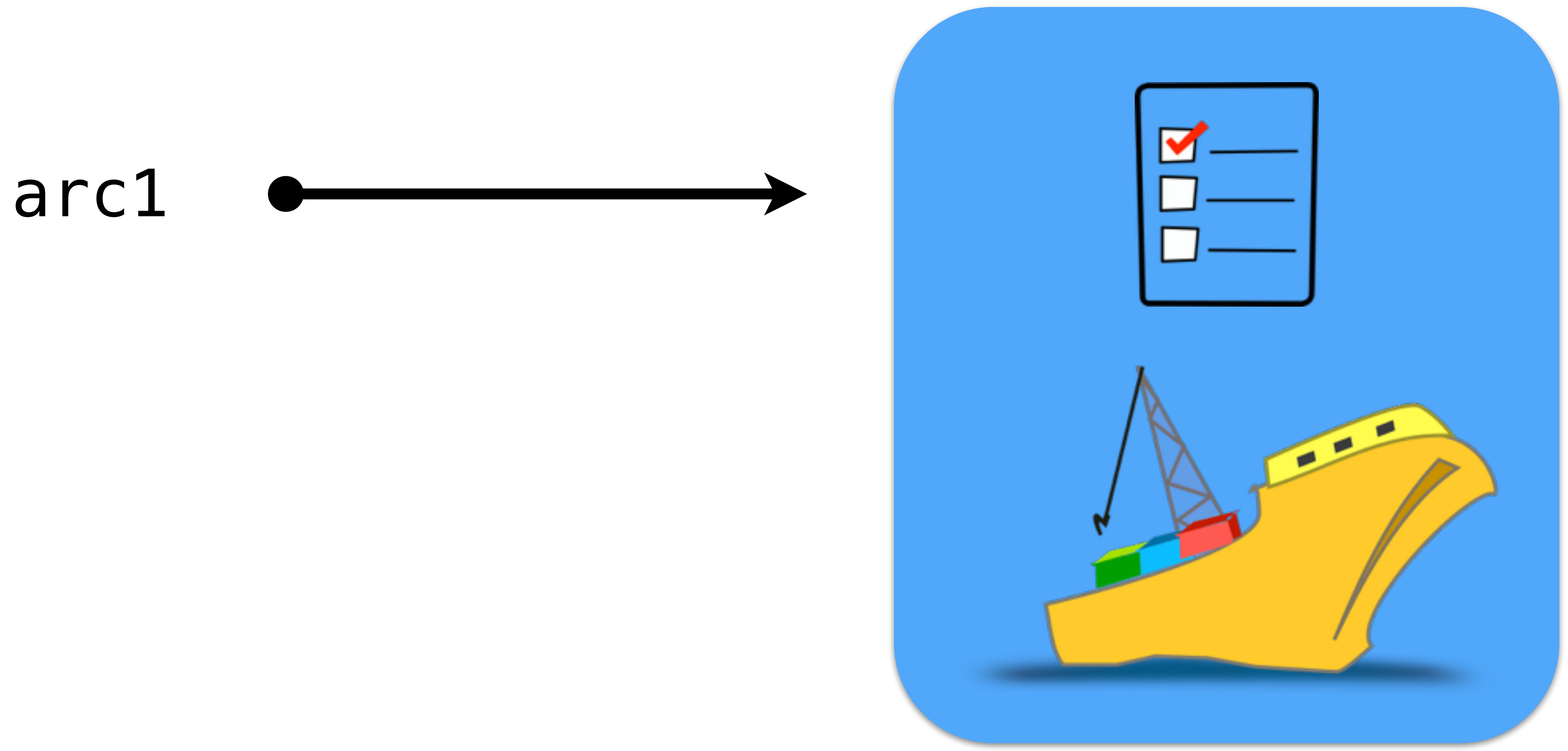
```rust
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```

```
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```

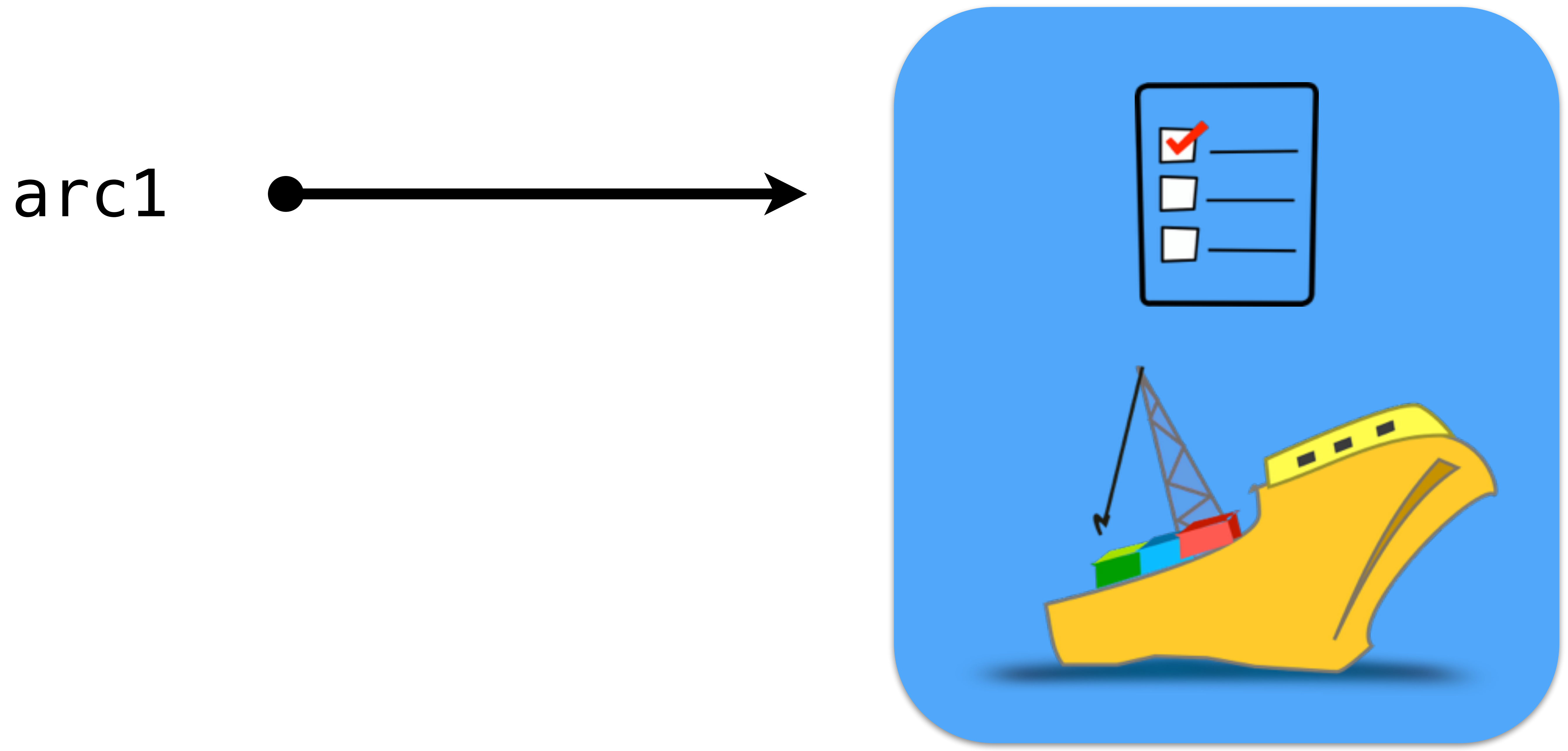
```rust
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```



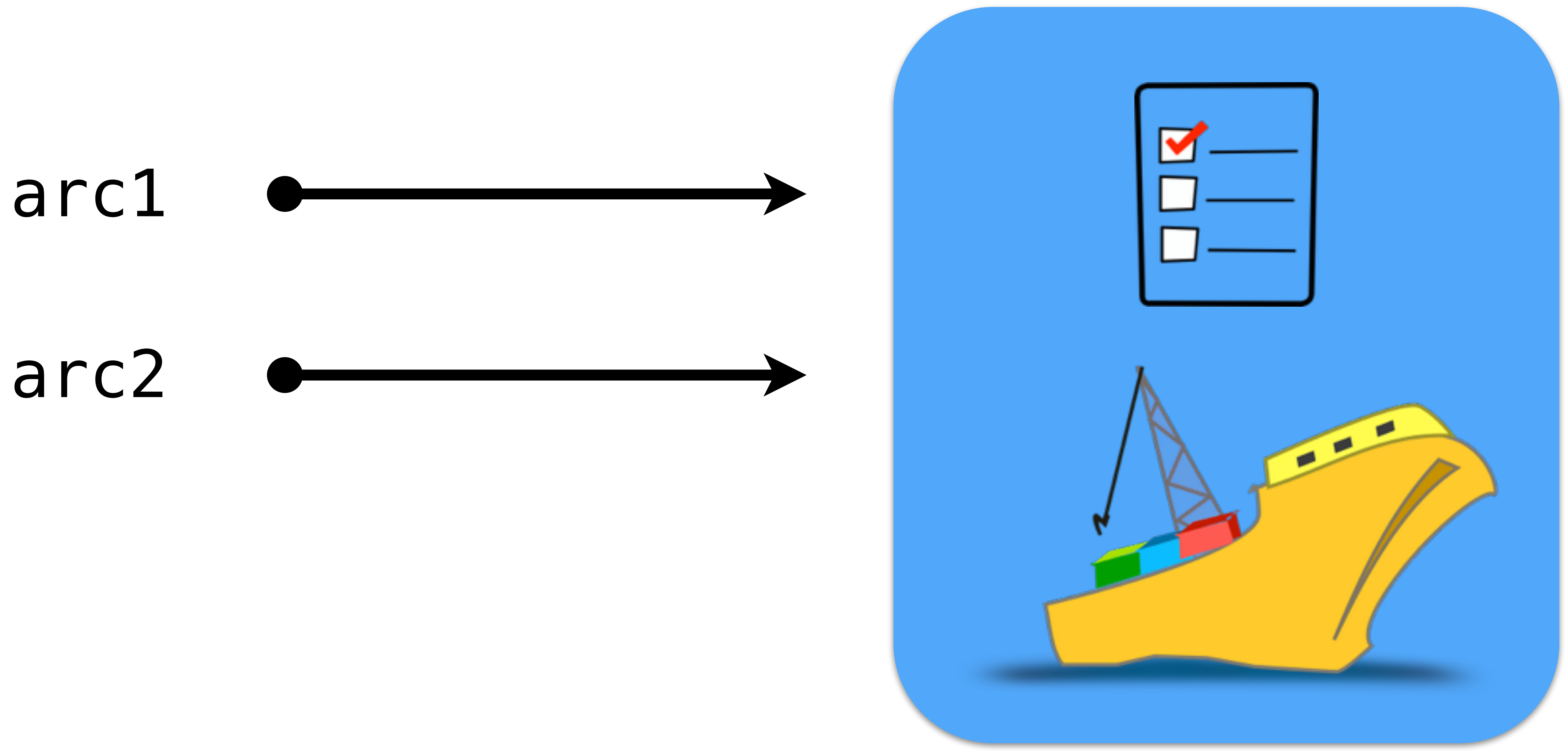
```rust
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```



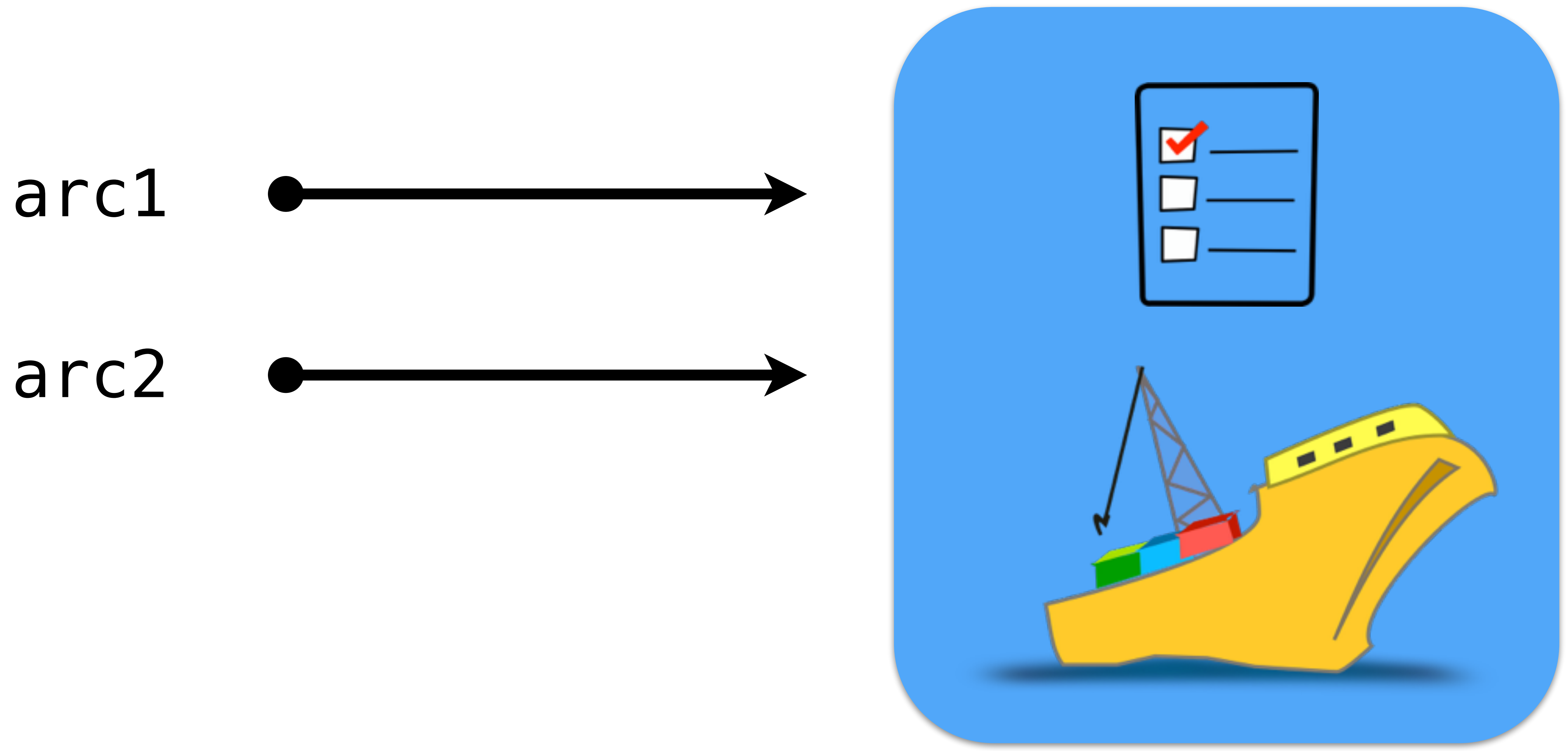
```
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```



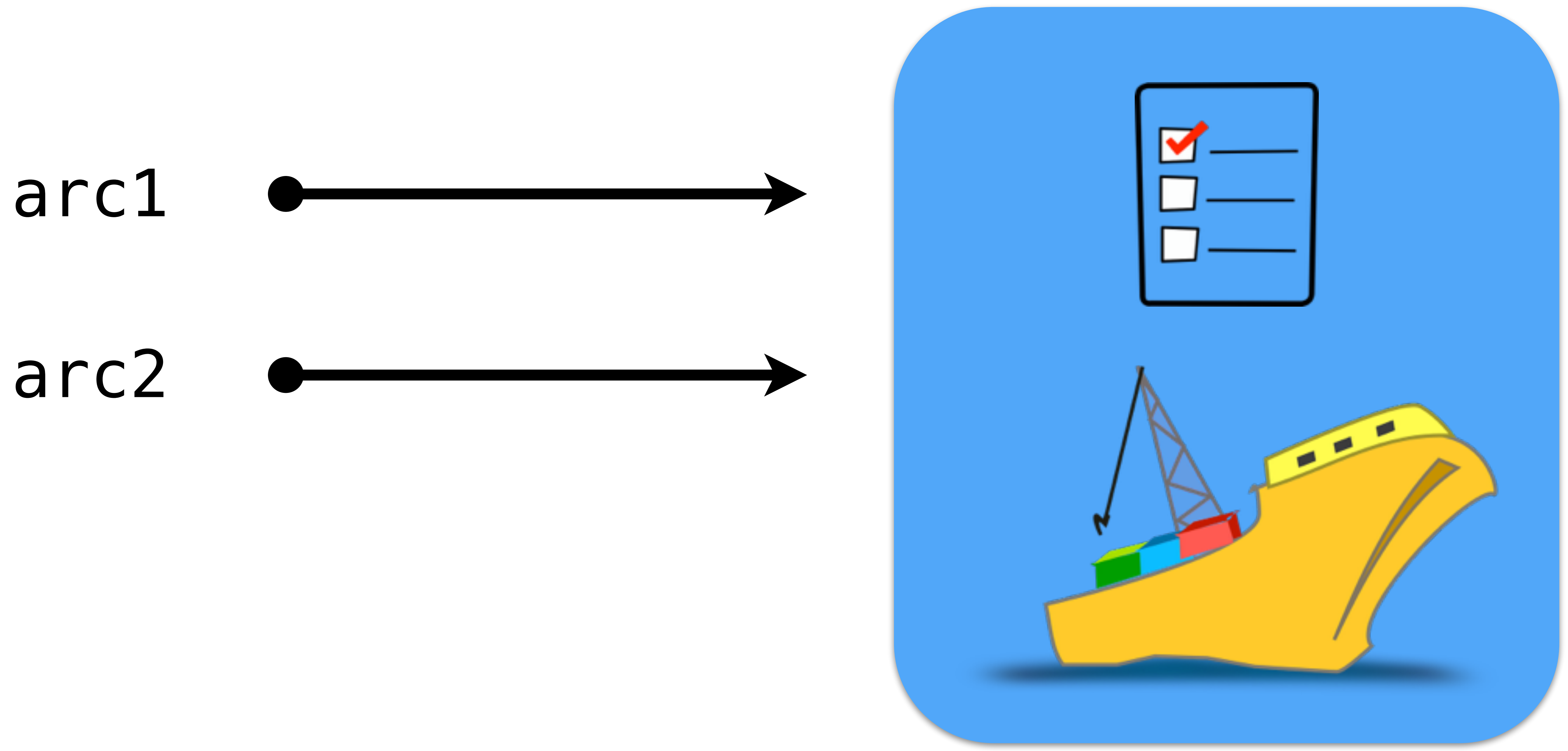
```
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```



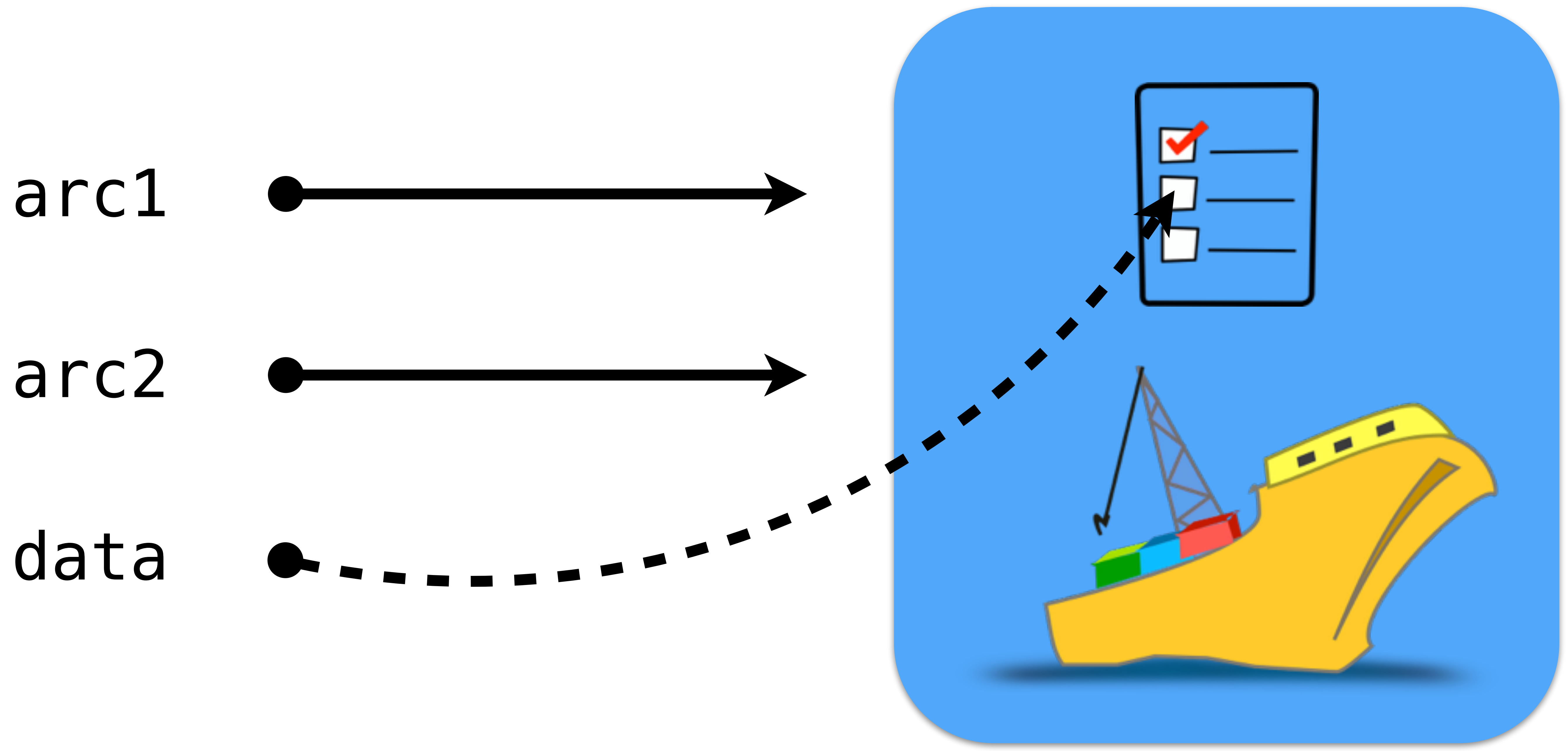
```
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```



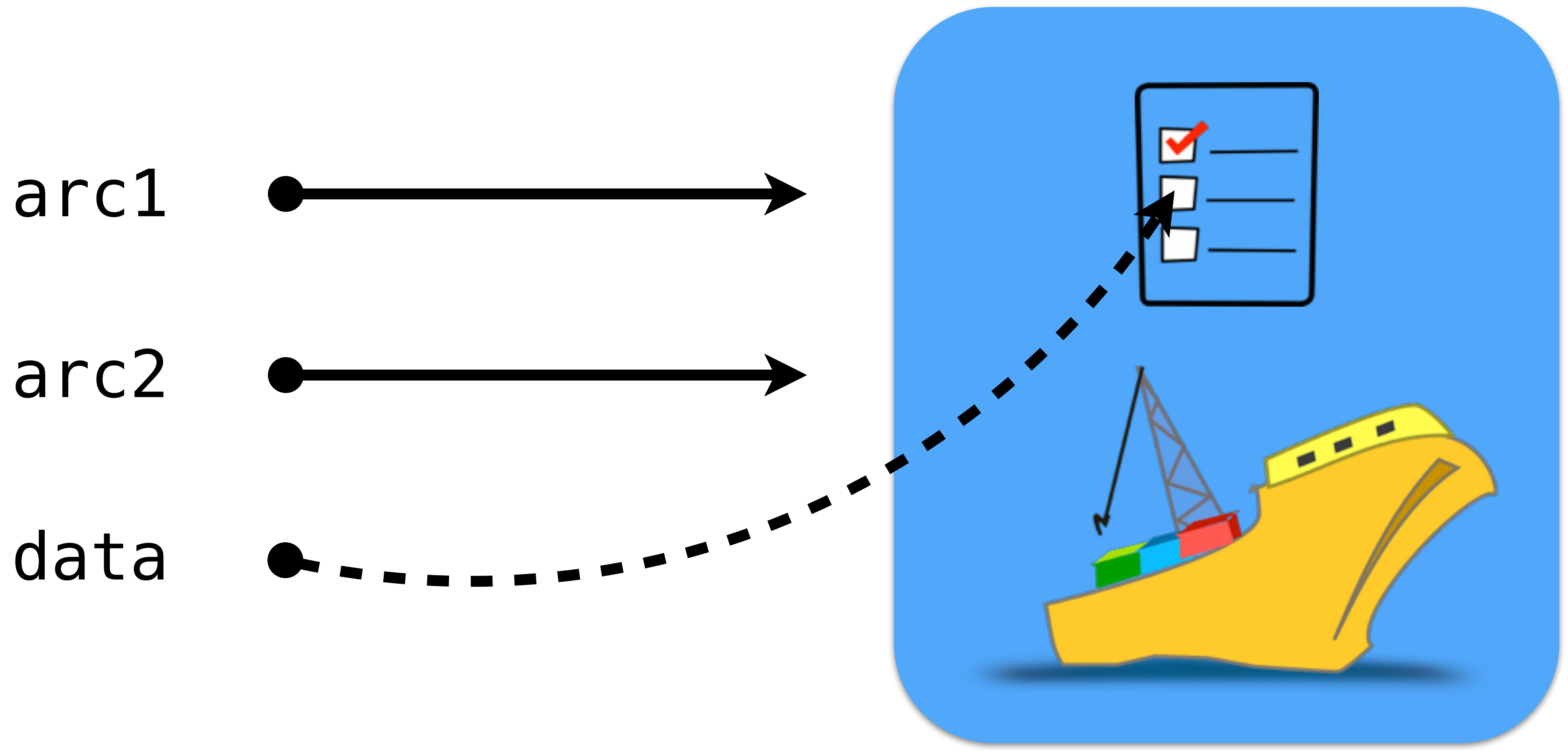
```
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```



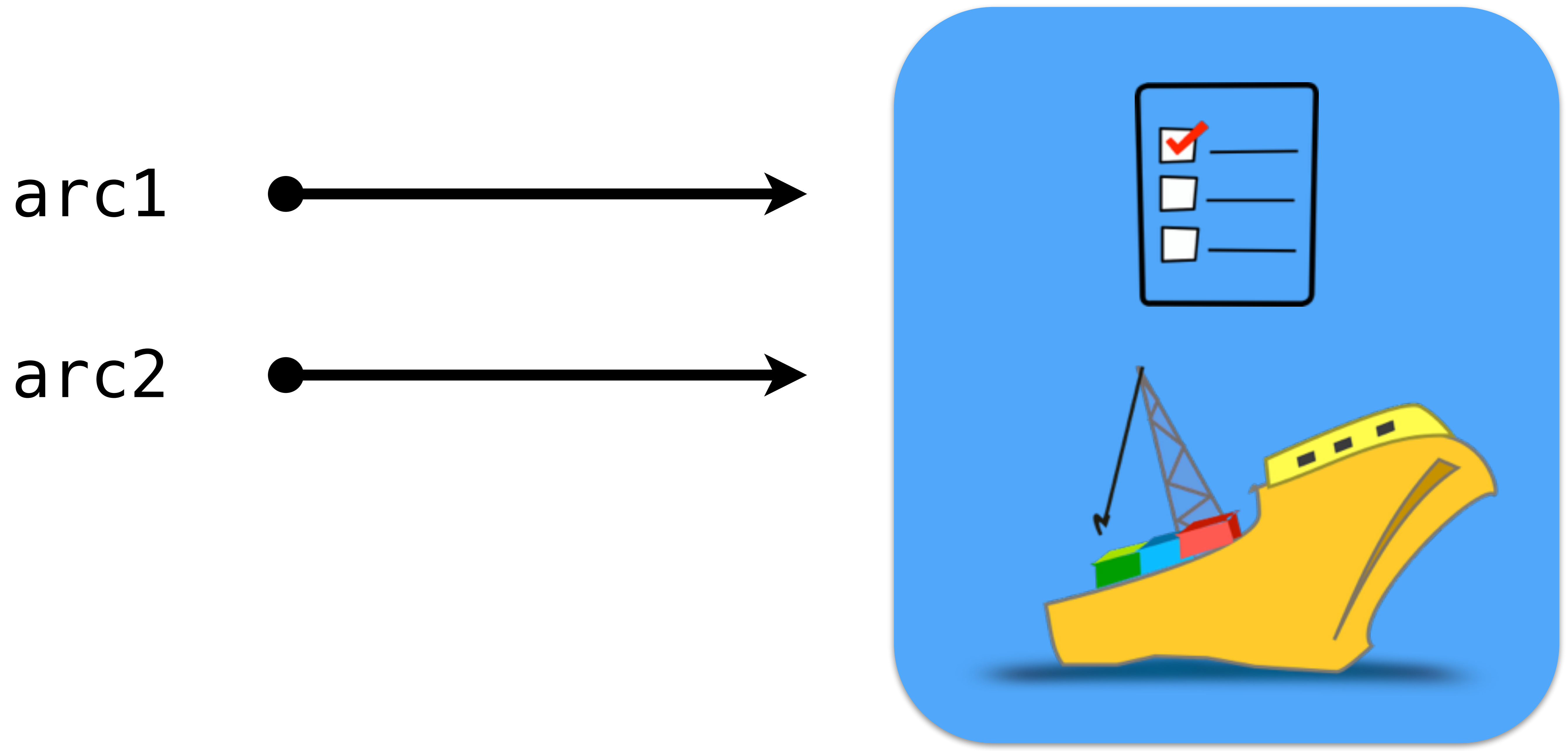
```
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```



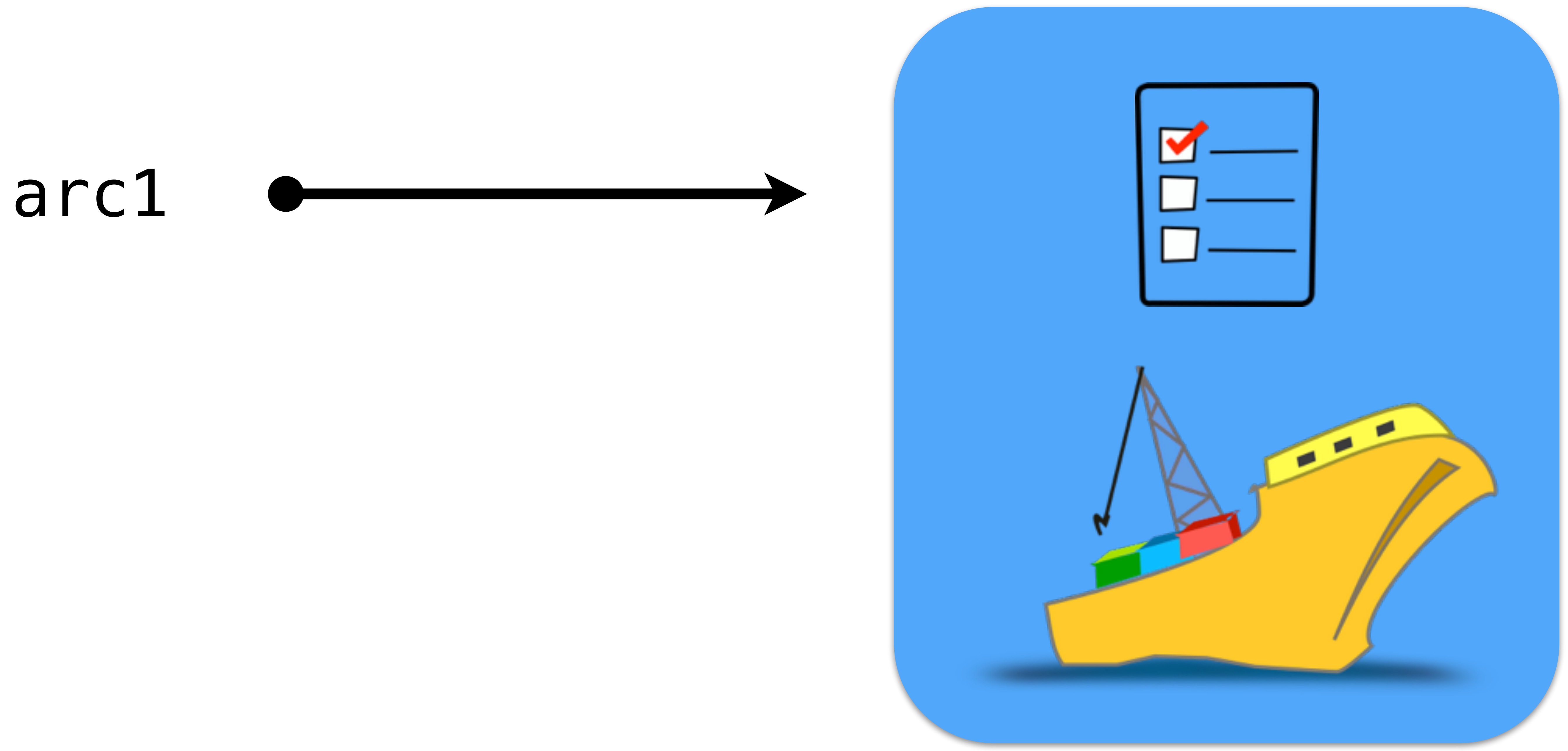
```rust
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```



```
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```



```rust
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```



```rust
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```

```rust
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let arc2 = arc1.clone();
let data = &arc1[0];
```

Arc => Immutable

```
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let data = &mut arc1[0];
```

Arc => Immutable

```
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let data = &mut arc1[0];
```

Arc => Immutable

```
use std::sync::Arc;
let shopping_list: Vec<ShoppingList> = …;
let arc1 = Arc::new(shopping_list);
let data = &mut arc1[0];
```

```
<anon>:6:21: 6:24 error: cannot borrow immutable borrowed
                         content as mutable
<anon>:6     let data = &mut arc[0];
                             ^~~
```

http://is.gd/nP3Pvb

```rust
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```rust
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```rust
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```rust
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

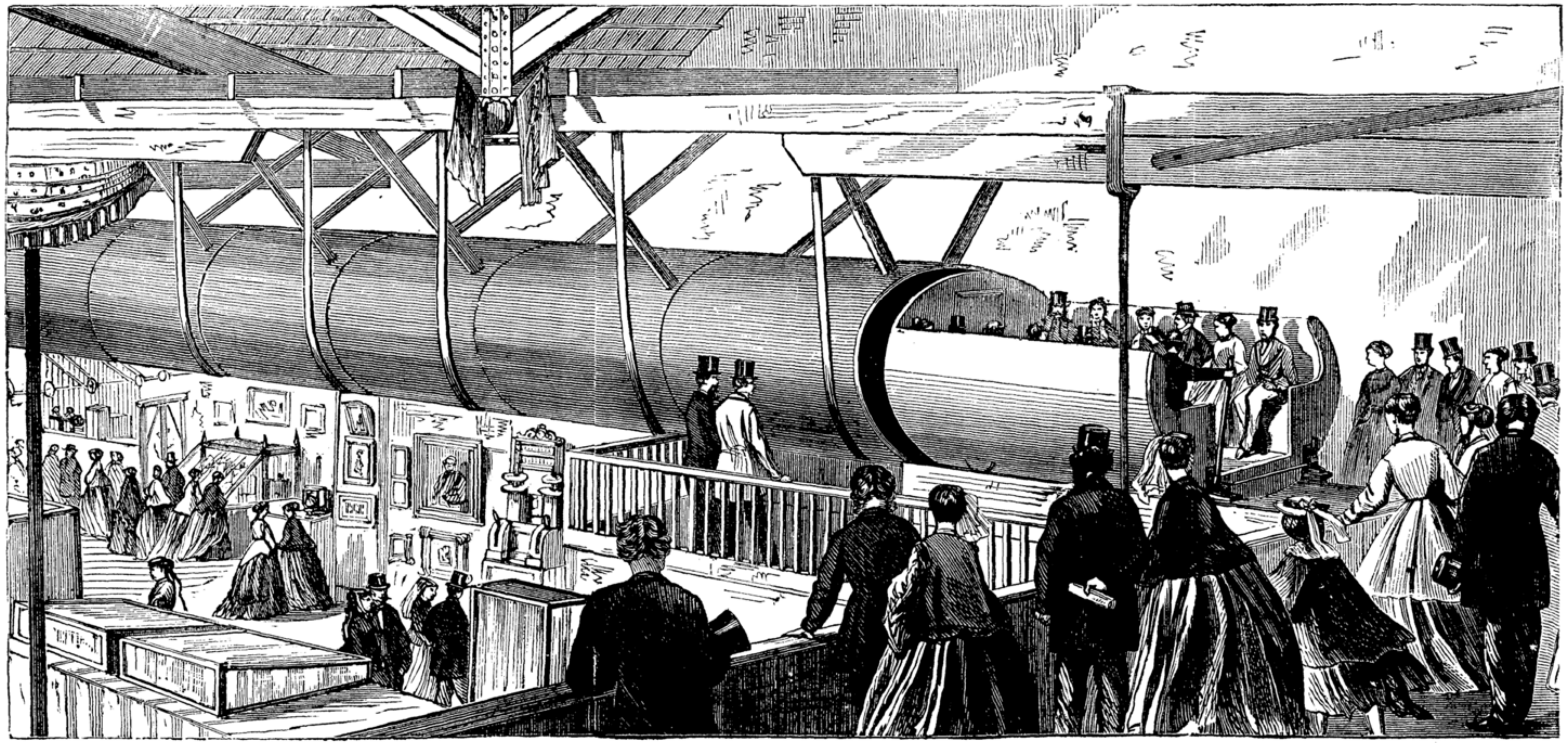
```
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}
```

```
use std::thread;
…
let shopping_list = Arc::new(vec![…]);
let mut handles = vec![];
for store in stores {
  let shopping_list = shopping_list.clone();
  let handle =
    thread::spawn(move || {
      let sum = store.total_price(shopping_list);
      (store.name, sum)
    });
  handles.push(handle);
}


// exercise (in a bit): join the handles!
```

Channels

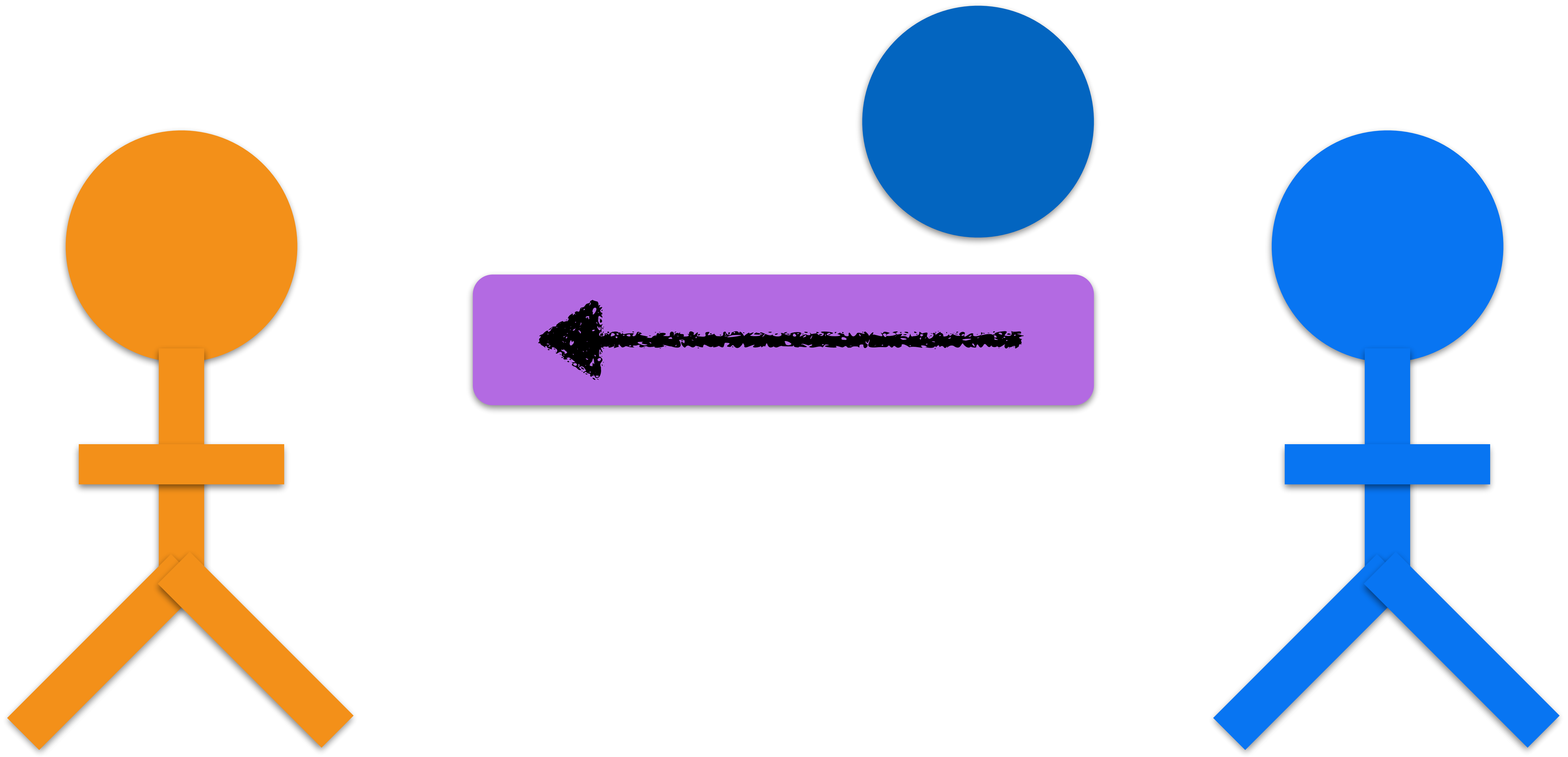


Joining a thread allows
thread to send **one result.**

What if we wanted
multiple results?

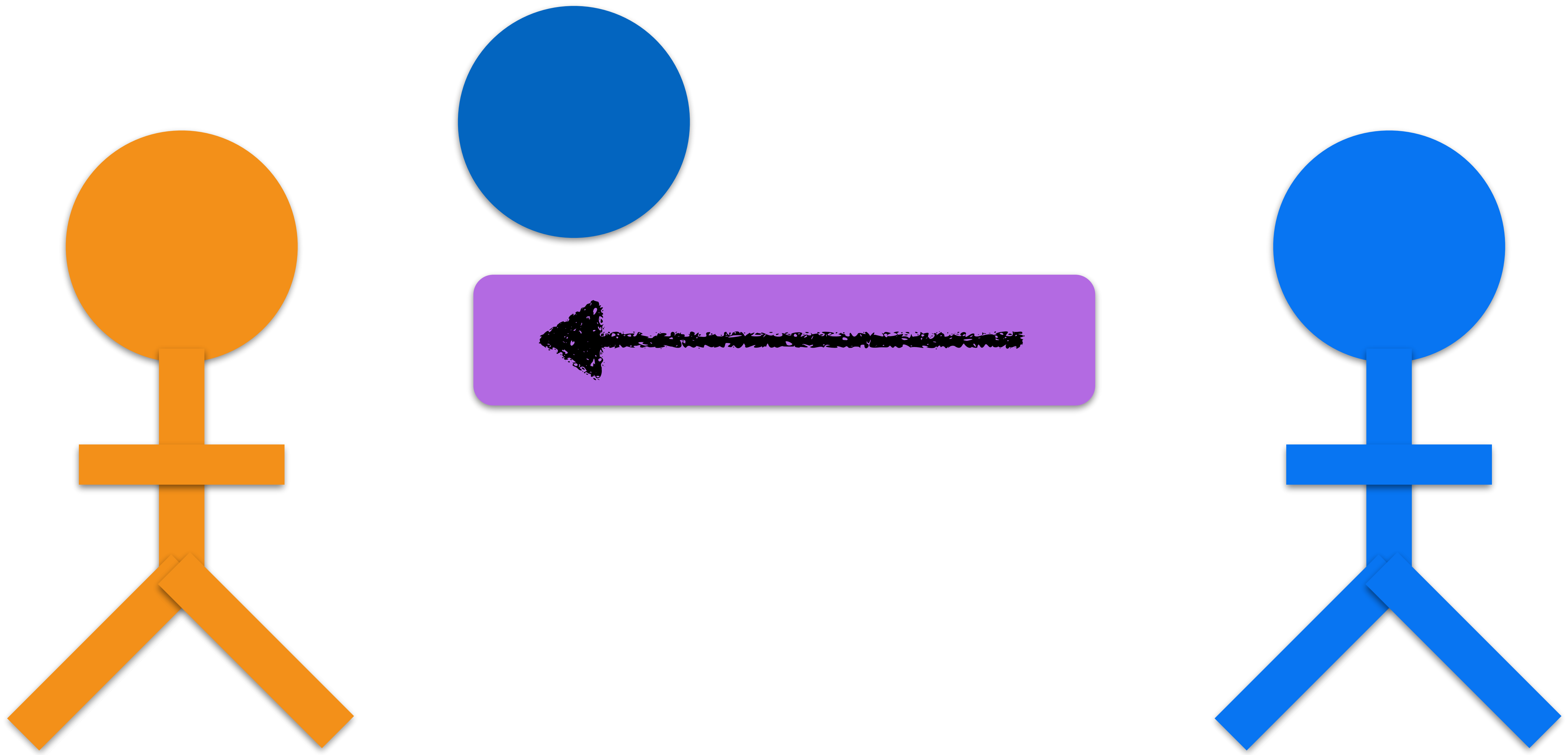
Joining a thread allows
thread to send **one result.**

What if we wanted
multiple results?



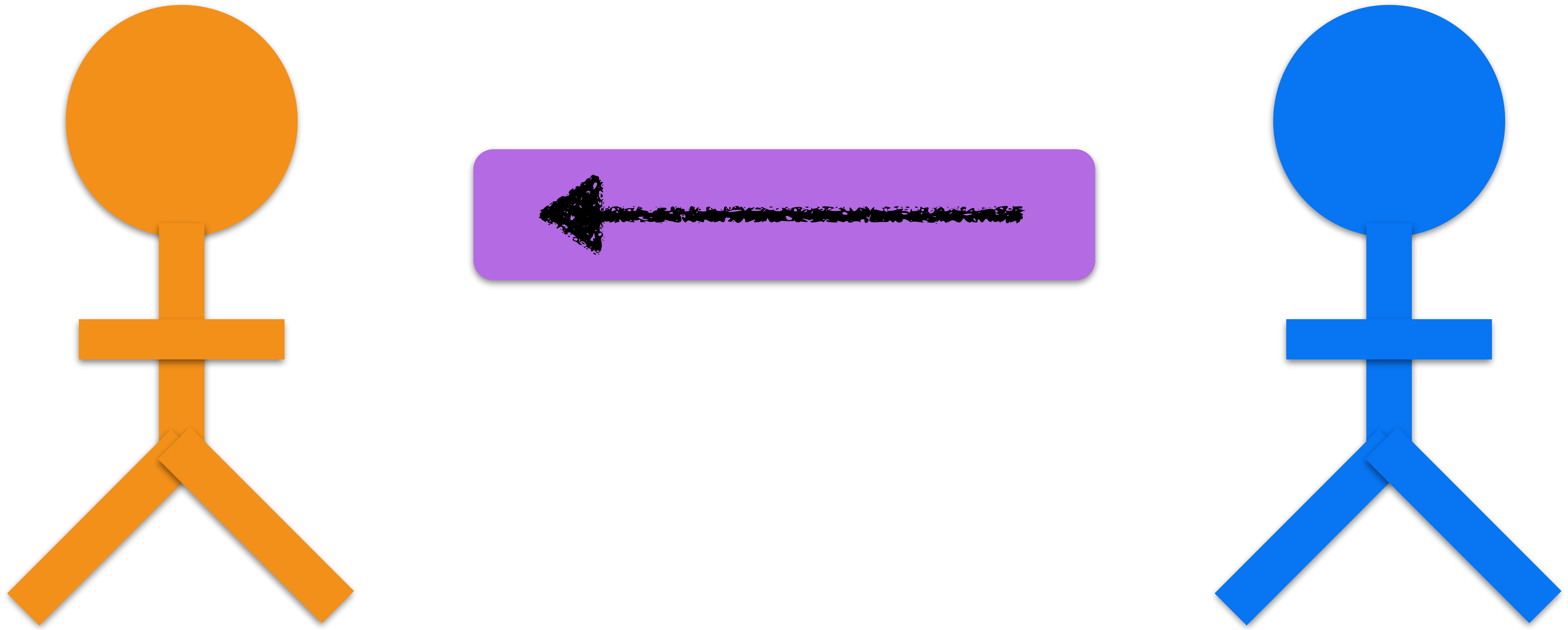
Joining a thread allows
thread to send **one result.**

What if we wanted
multiple results?



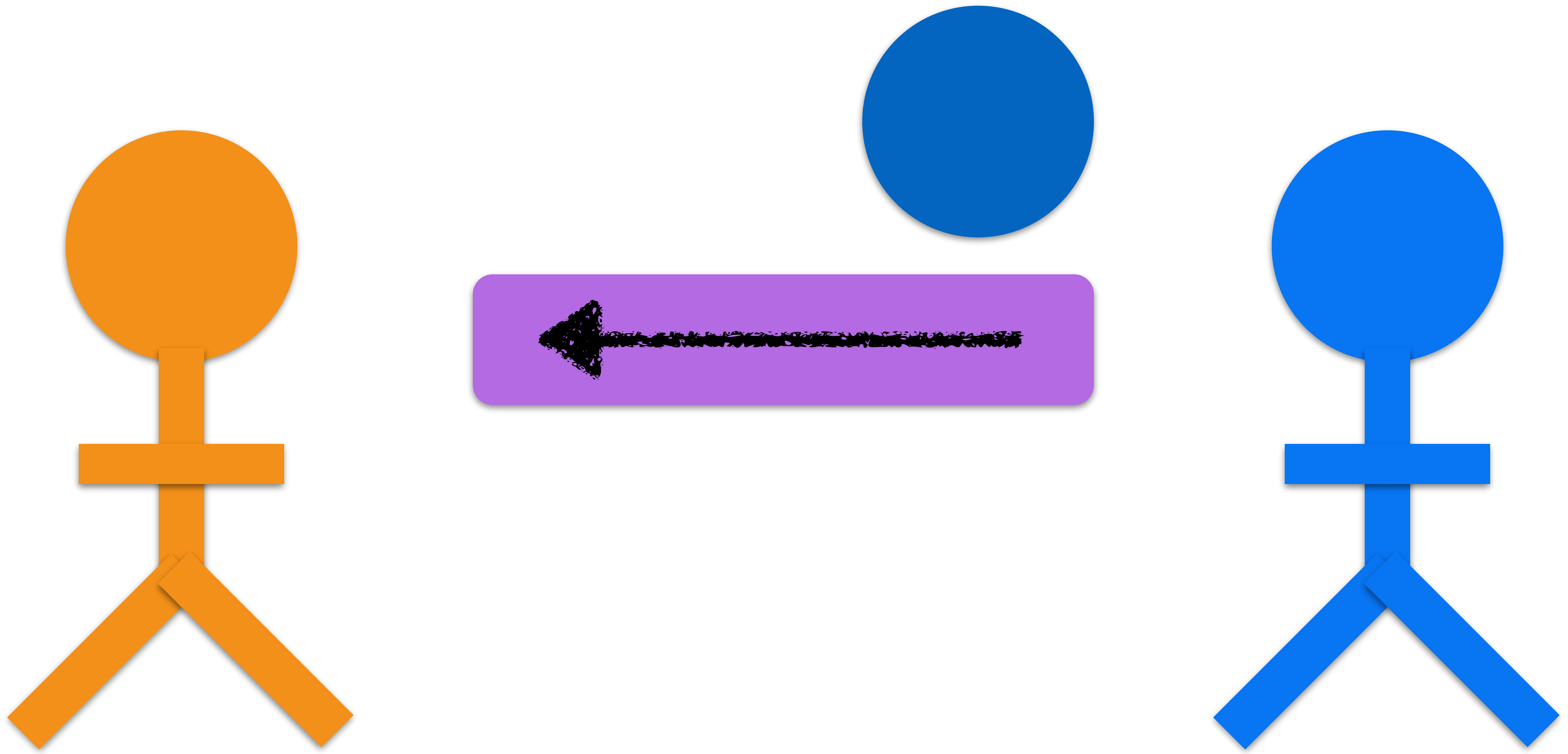
Joining a thread allows thread to send **one result.**

What if we wanted **multiple results?**



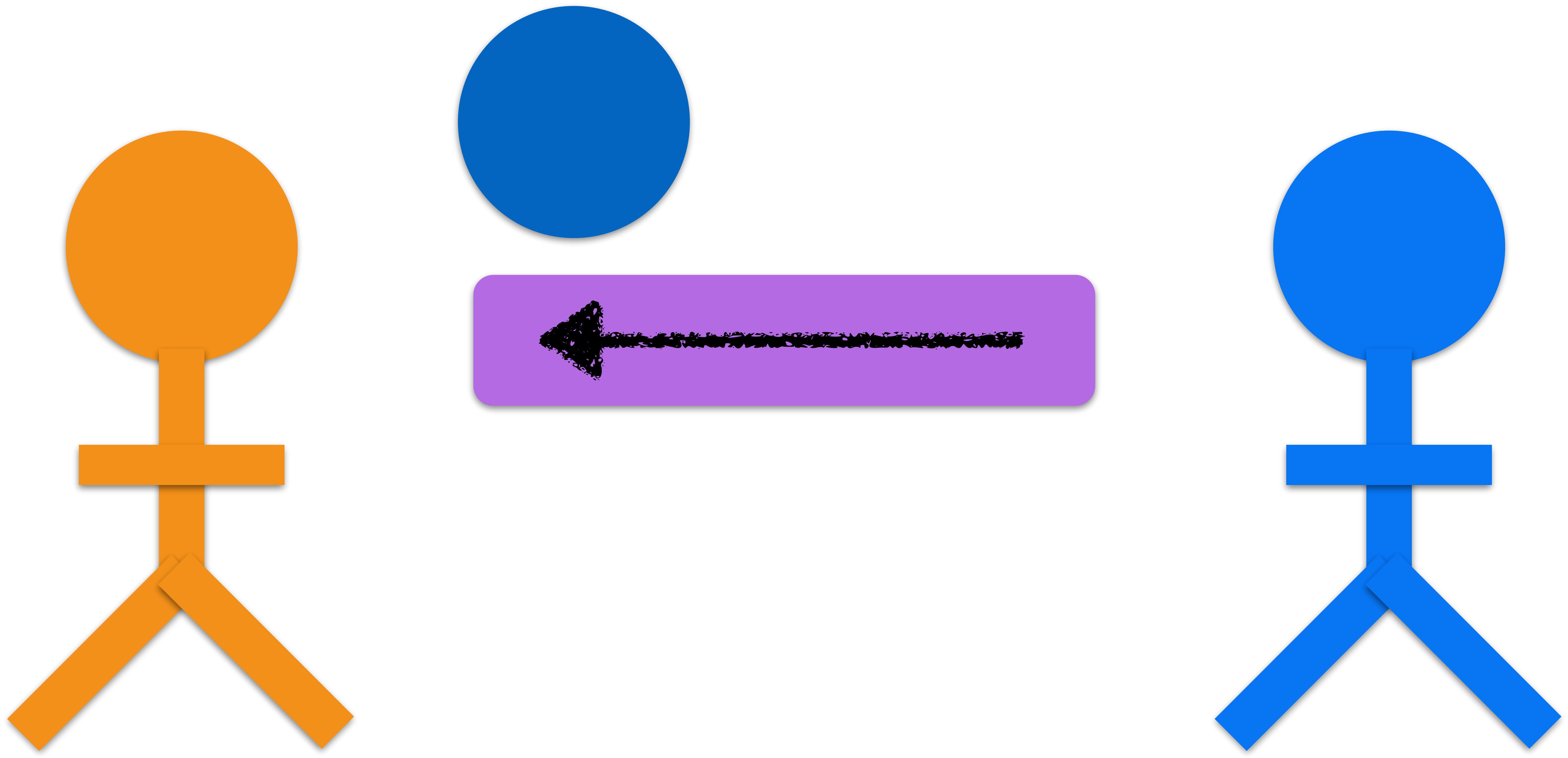
Joining a thread allows thread to send **one result.**

What if we wanted **multiple results?**



Joining a thread allows thread to send **one result.**

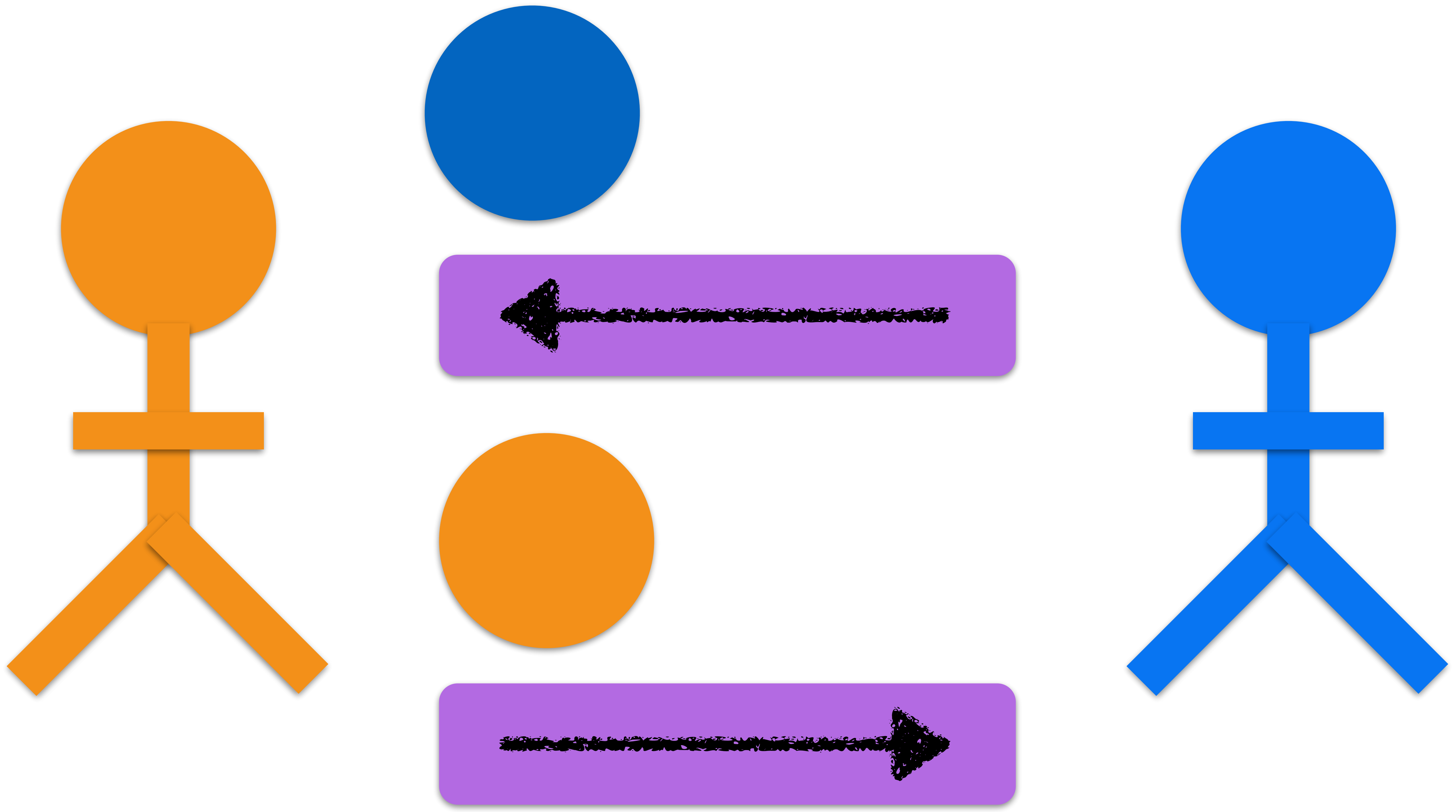
What if we wanted **multiple results?**



Joining a thread allows thread to send **one result.**

What if we wanted **multiple results?**

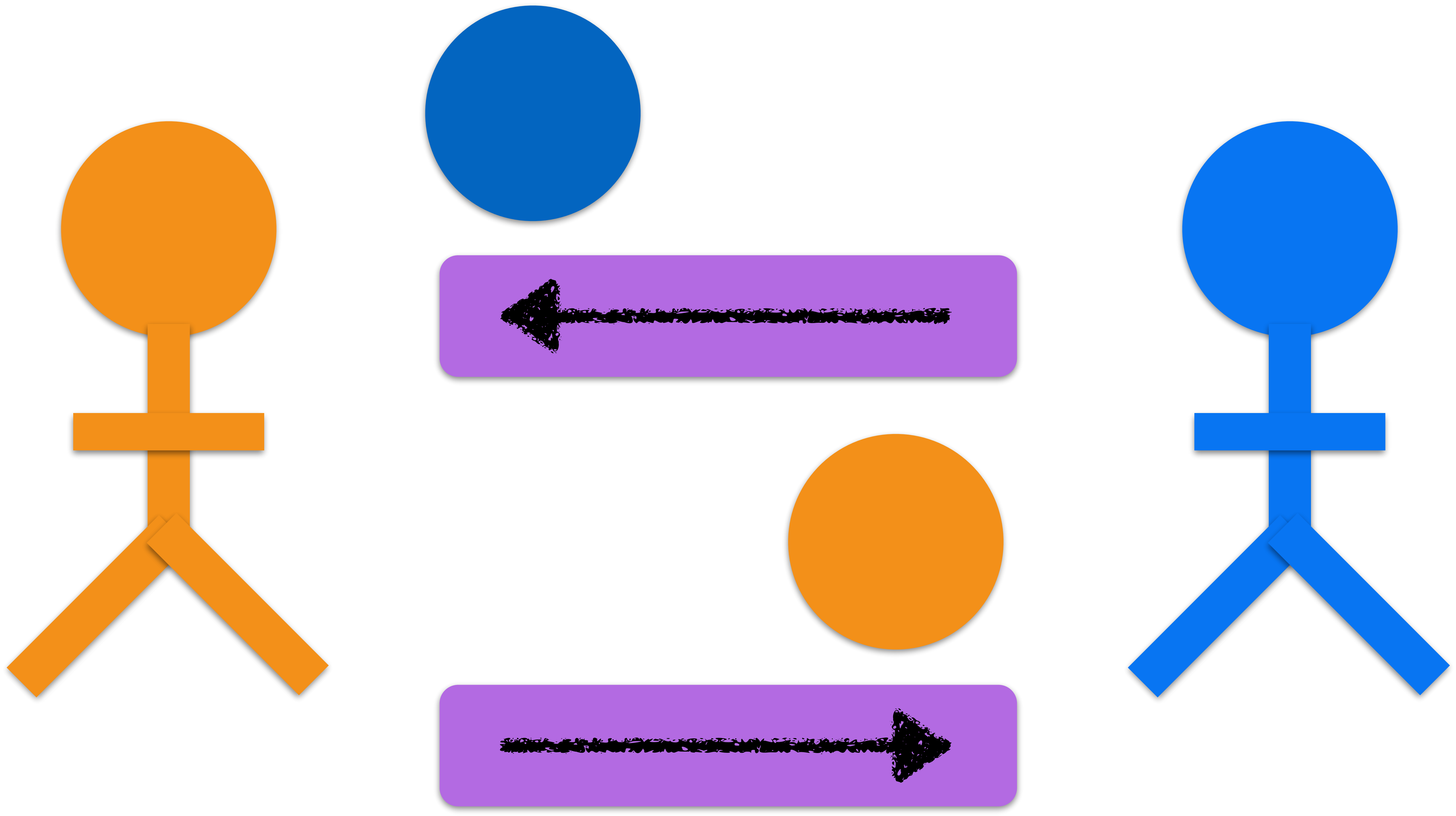
Or if we wanted a **response?**



Joining a thread allows
thread to send **one result.**

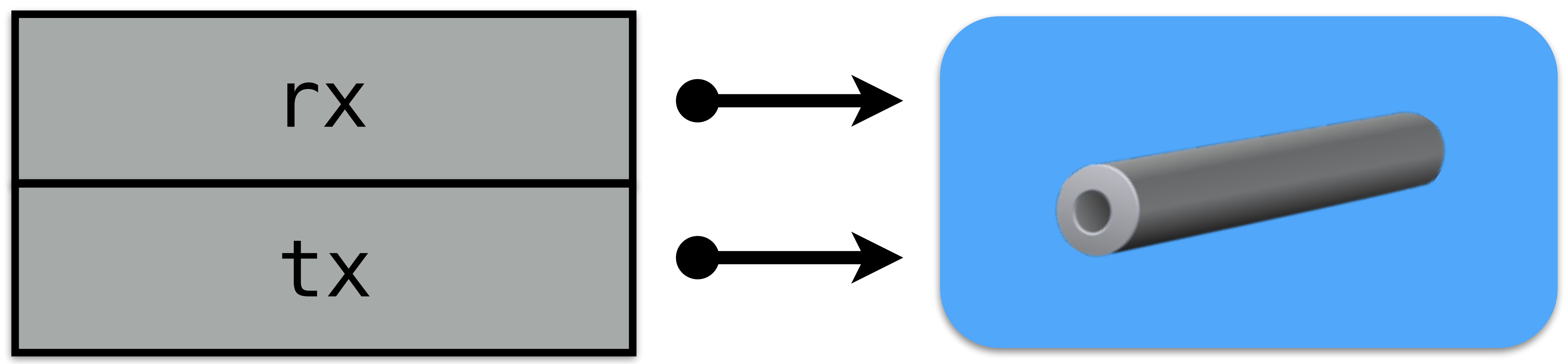
What if we wanted
multiple results?

Or if we wanted
a **response?**

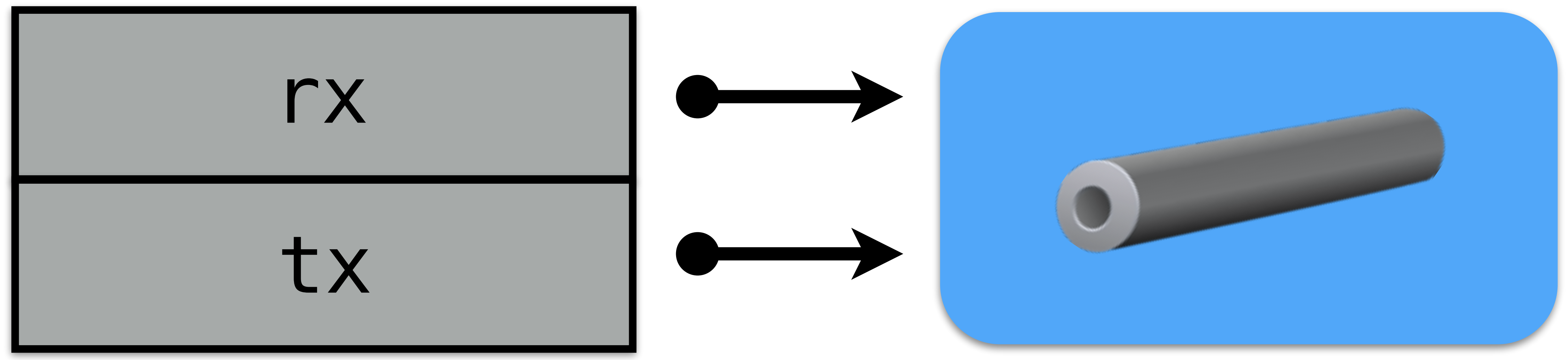


```
fn parent() {
  let (tx, rx) = channel();
  spawn(move || {…});
  let m = rx.recv().unwrap();
}
```

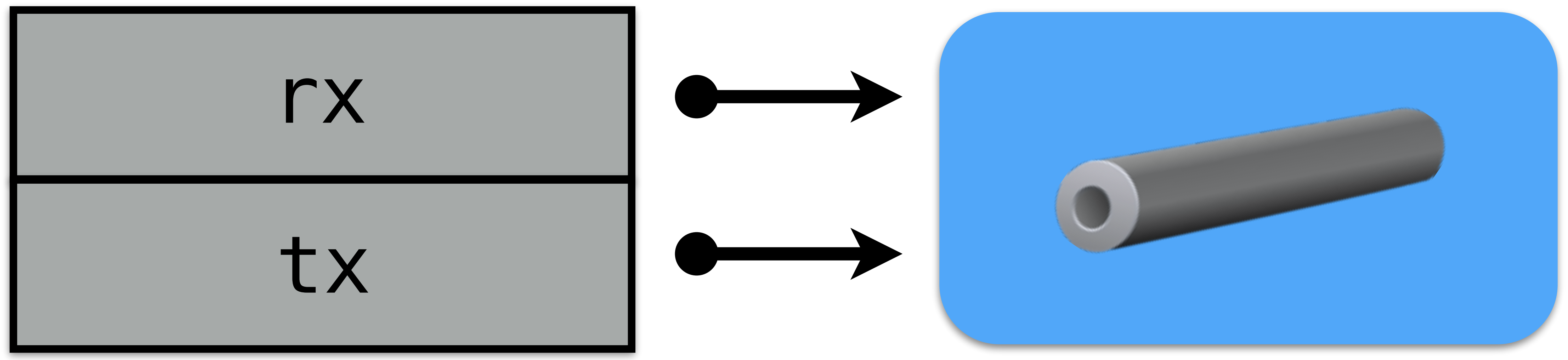
```
fn parent() {
  let (tx, rx) = channel();
  spawn(move || {…});
  let m = rx.recv().unwrap();
}
```



```
fn parent() {
  let (tx, rx) = channel();
  spawn(move || {…});
  let m = rx.recv().unwrap();
}
```

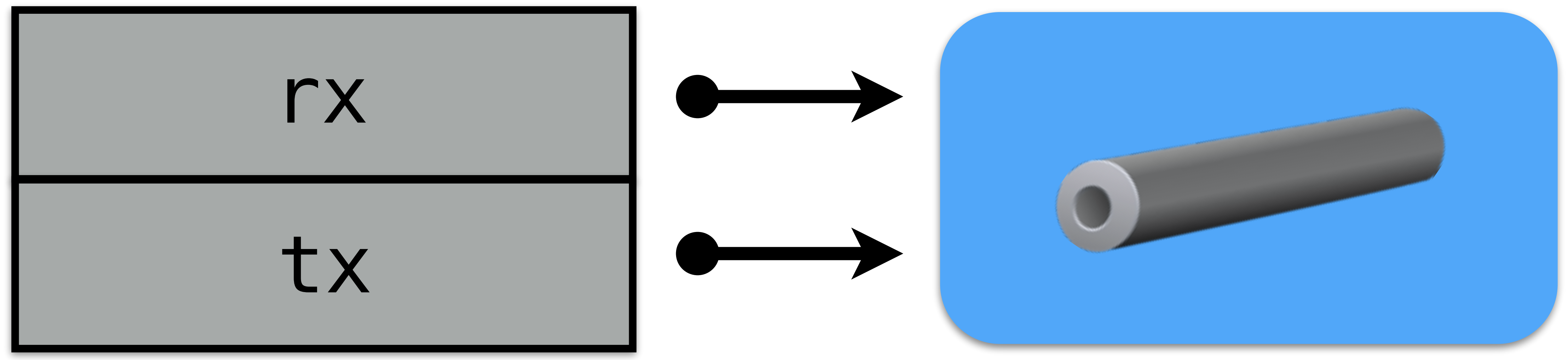


```
fn parent() {
  let (tx, rx) = channel();
  spawn(move || {…});
  let m = rx.recv().unwrap();
}
```



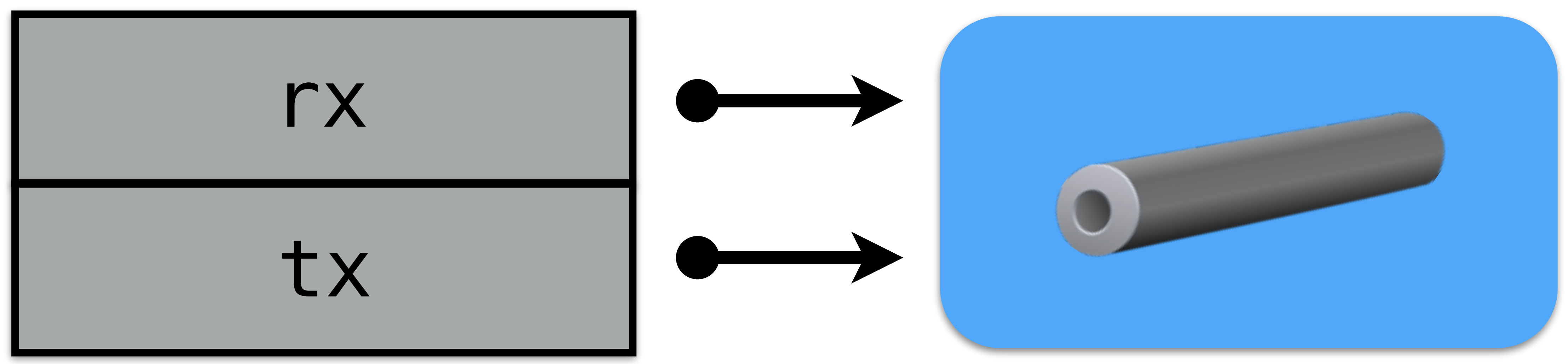
```
fn parent() {
  let (tx, rx) = channel();
  spawn(move || {…});
  let m = rx.recv().unwrap();
}
```

```
move || {
  let m = Message::new();
  …
  tx.send(m).unwrap();
}
```

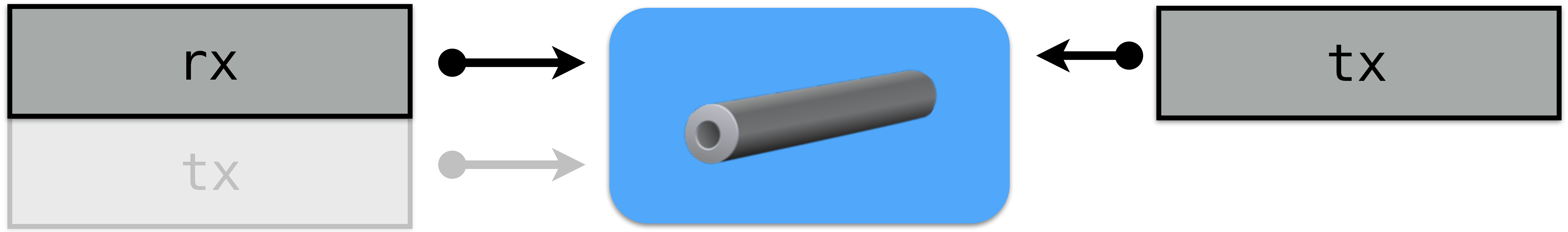


```
fn parent() {
  let (tx, rx) = channel();
  spawn(move || {…});
  let m = rx.recv().unwrap();
}
```

```
move || {
  let m = Message::new();
  …
  tx.send(m).unwrap();
}
```

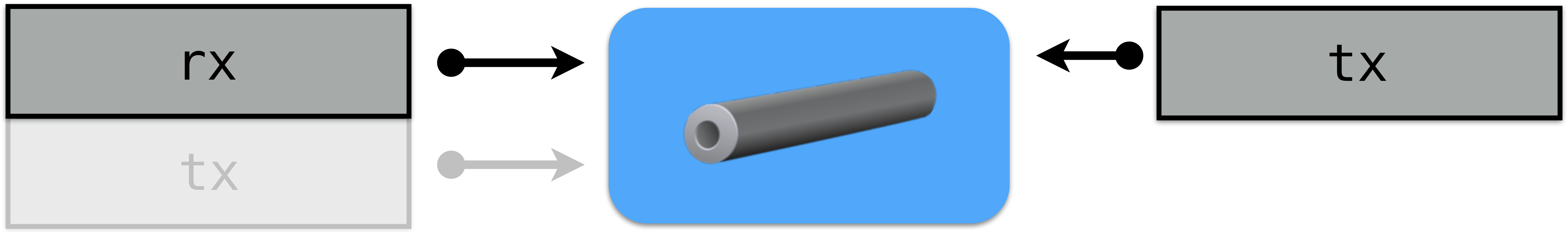


```
fn parent() {                          move || {
  let (tx, rx) = channel();              let m = Message::new();
  spawn(move || {…});                    …
  let m = rx.recv().unwrap();            tx.send(m).unwrap();
}                                      }
```



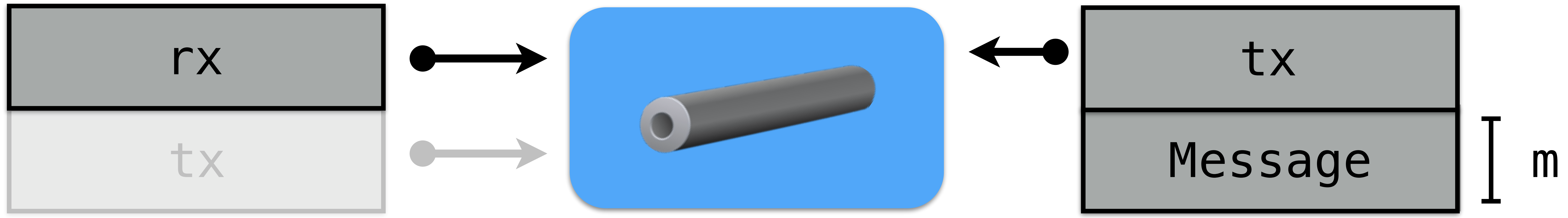
```
fn parent() {
  let (tx, rx) = channel();
  spawn(move || {…});
  let m = rx.recv().unwrap();
}
```

```
move || {
  let m = Message::new();
  …
  tx.send(m).unwrap();
}
```

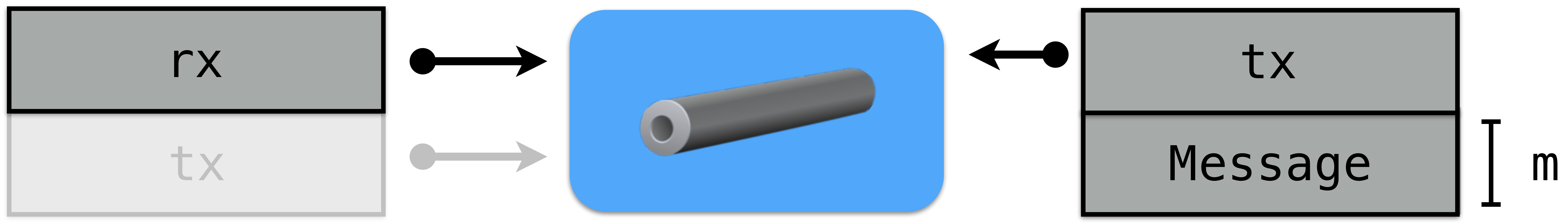


```
fn parent() {
  let (tx, rx) = channel();
  spawn(move || {…});
  let m = rx.recv().unwrap();
}
```

```
move || {
  let m = Message::new();
  …
  tx.send(m).unwrap();
}
```

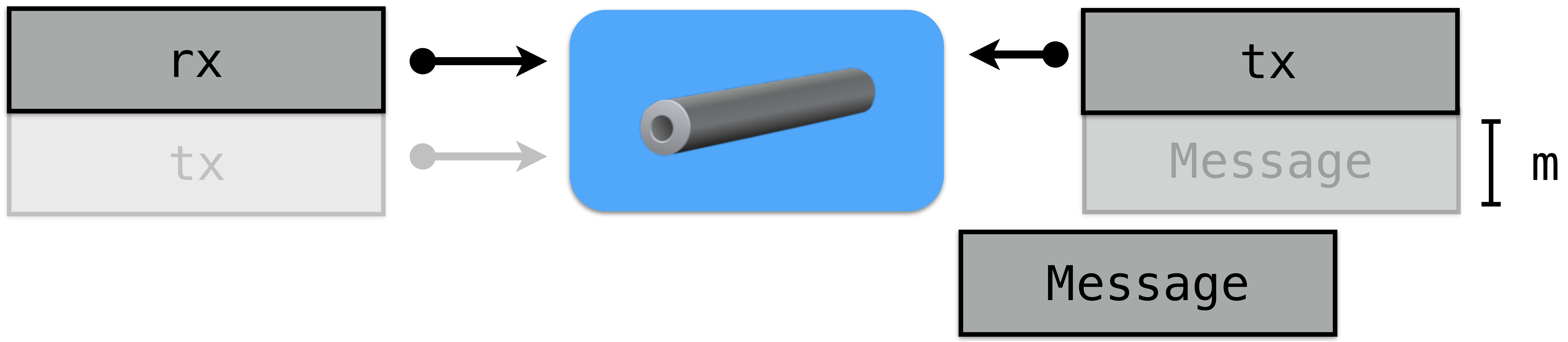


```
fn parent() {                          move || {
  let (tx, rx) = channel();              let m = Message::new();
  spawn(move || {…});                    …
  let m = rx.recv().unwrap();            tx.send(m).unwrap();
}                                      }
```



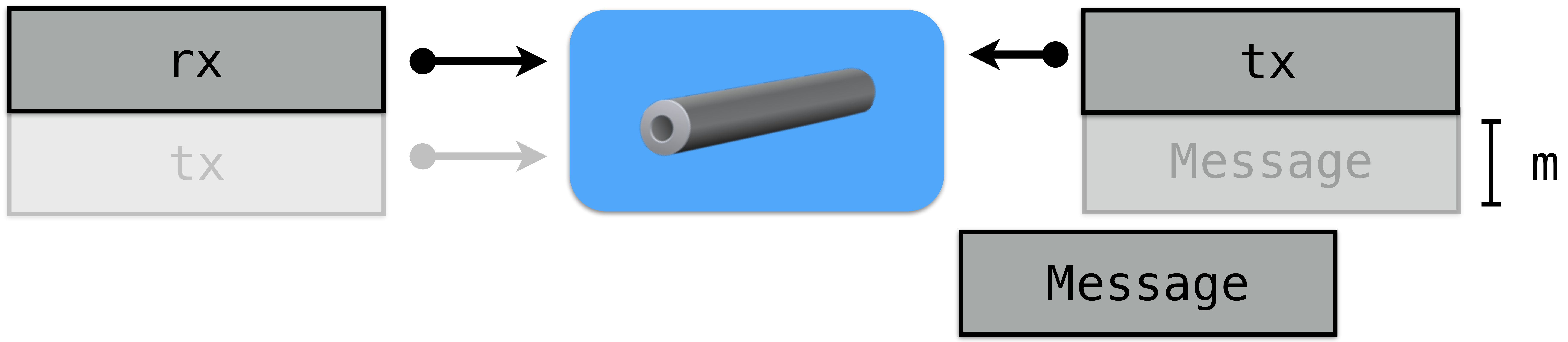
```
fn parent() {
  let (tx, rx) = channel();
  spawn(move || {…});
  let m = rx.recv().unwrap();
}
```

```
move || {
  let m = Message::new();
  …
  tx.send(m).unwrap();
}
```

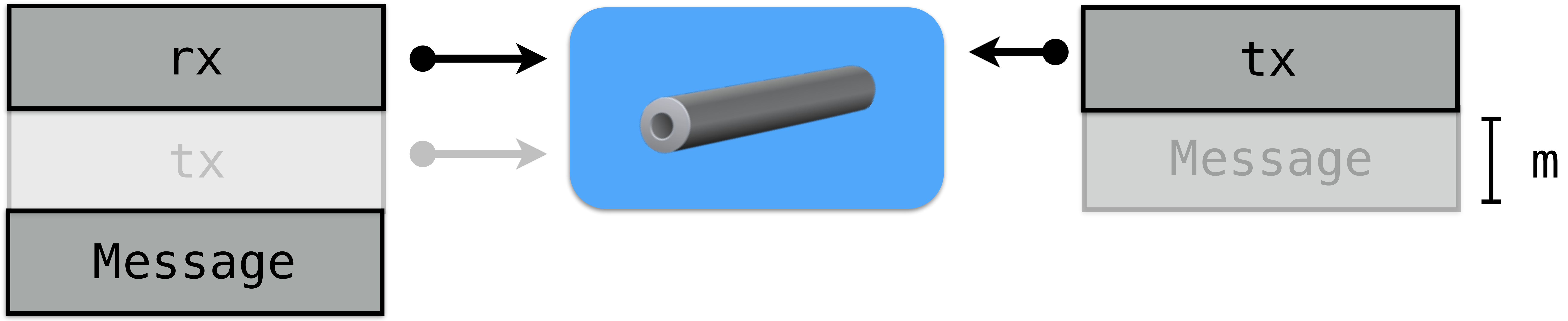


```
fn parent() {
  let (tx, rx) = channel();
  spawn(move || {…});
  let m = rx.recv().unwrap();
}
```

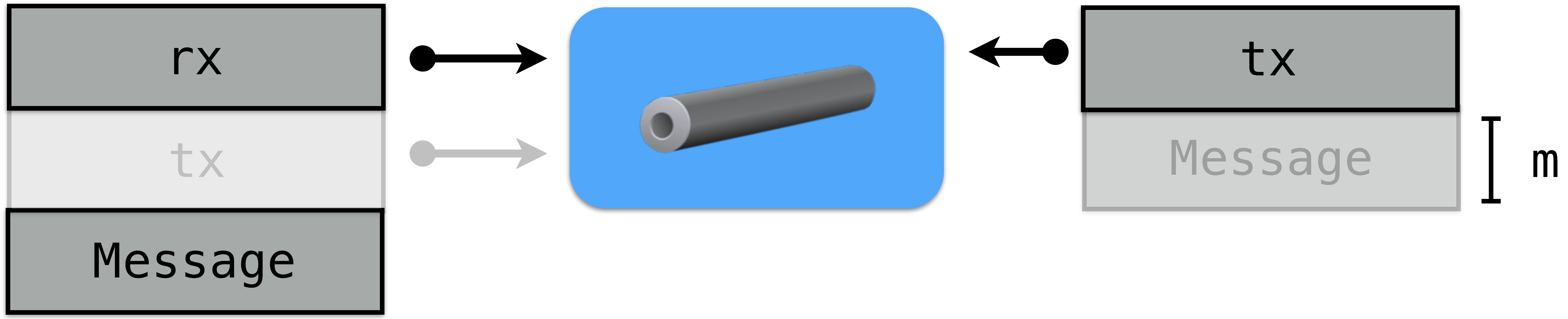
```
move || {
  let m = Message::new();
  …
  tx.send(m).unwrap();
}
```



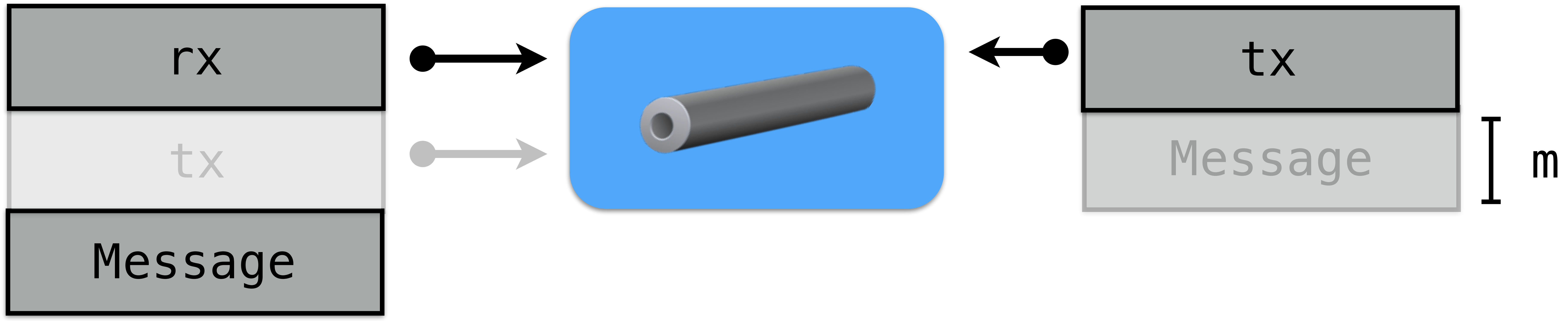
```
 fn parent() {                       move || {
   let (tx, rx) = channel();           let m = Message::new();
   spawn(move || {…});                 …
➡ let m = rx.recv().unwrap();          tx.send(m).unwrap();
 }                                   }
```



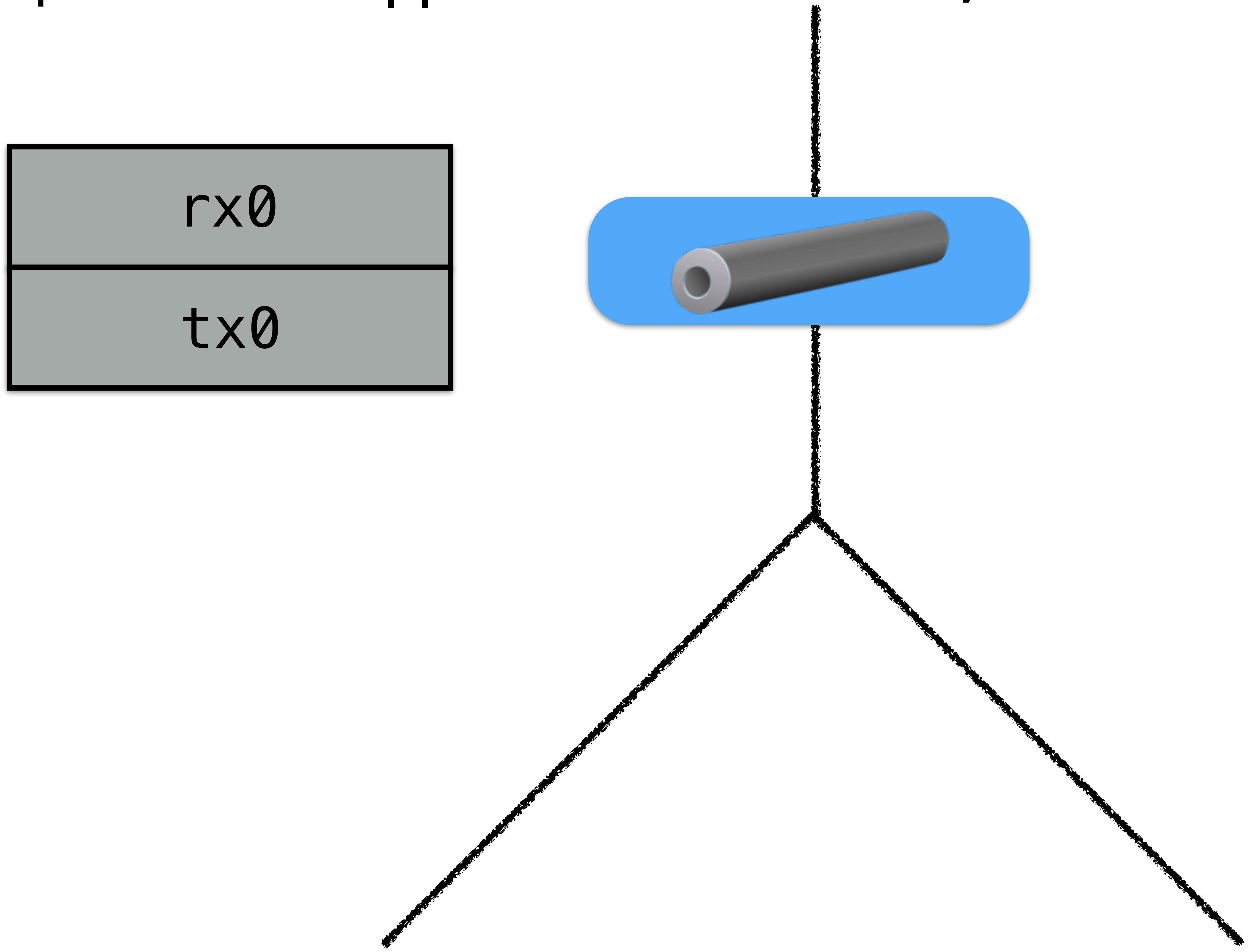
```
fn parent() {                             move || {
  let (tx, rx) = channel();                 let m = Message::new();
  spawn(move || {…});                       …
  let m = rx.recv().unwrap();               tx.send(m).unwrap();
}                                         }
```



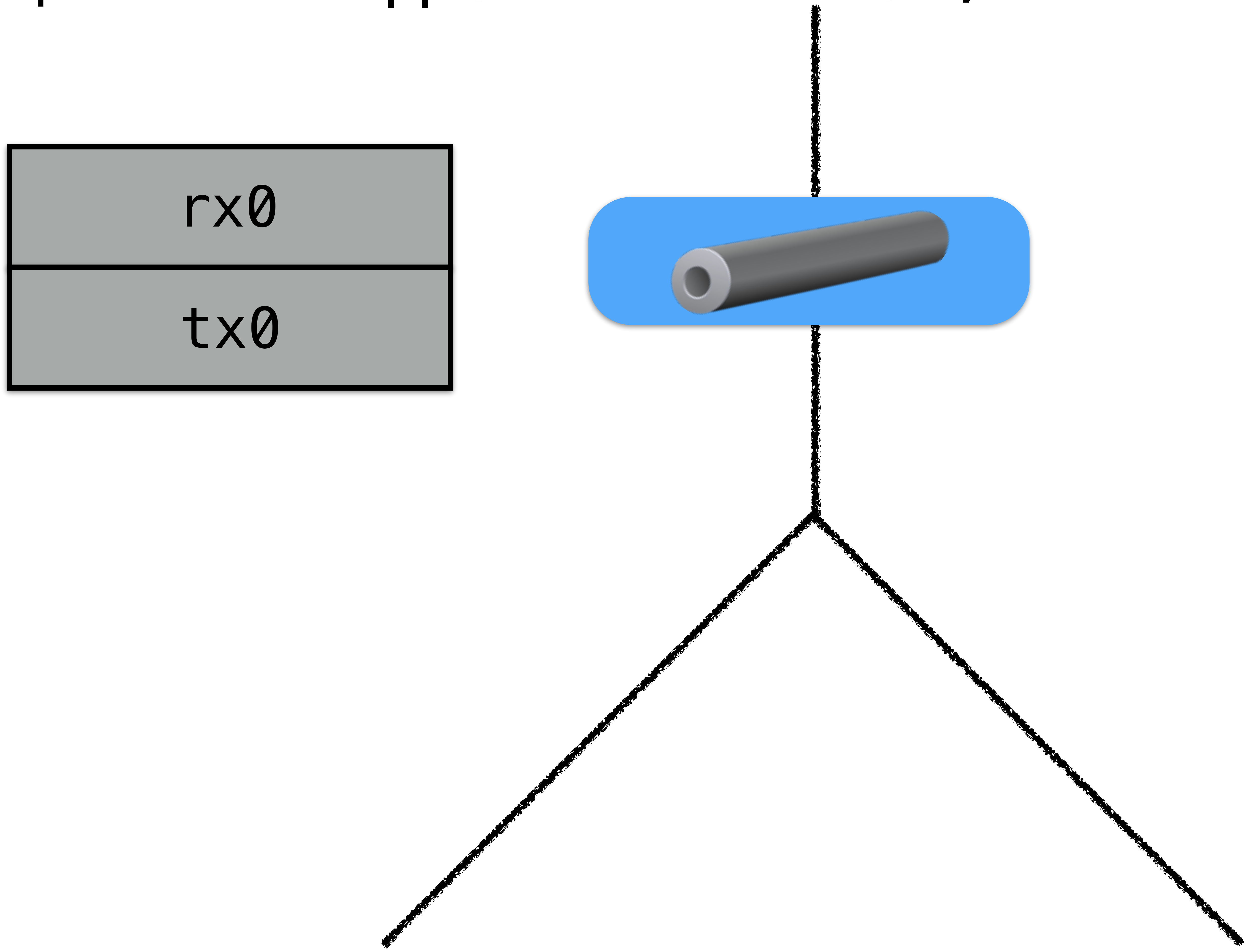
```
fn parent() {                         move || {
  let (tx, rx) = channel();             let m = Message::new();
  spawn(move || {…});                   …
  let m = rx.recv().unwrap();           tx.send(m).unwrap();
}                                     }
```



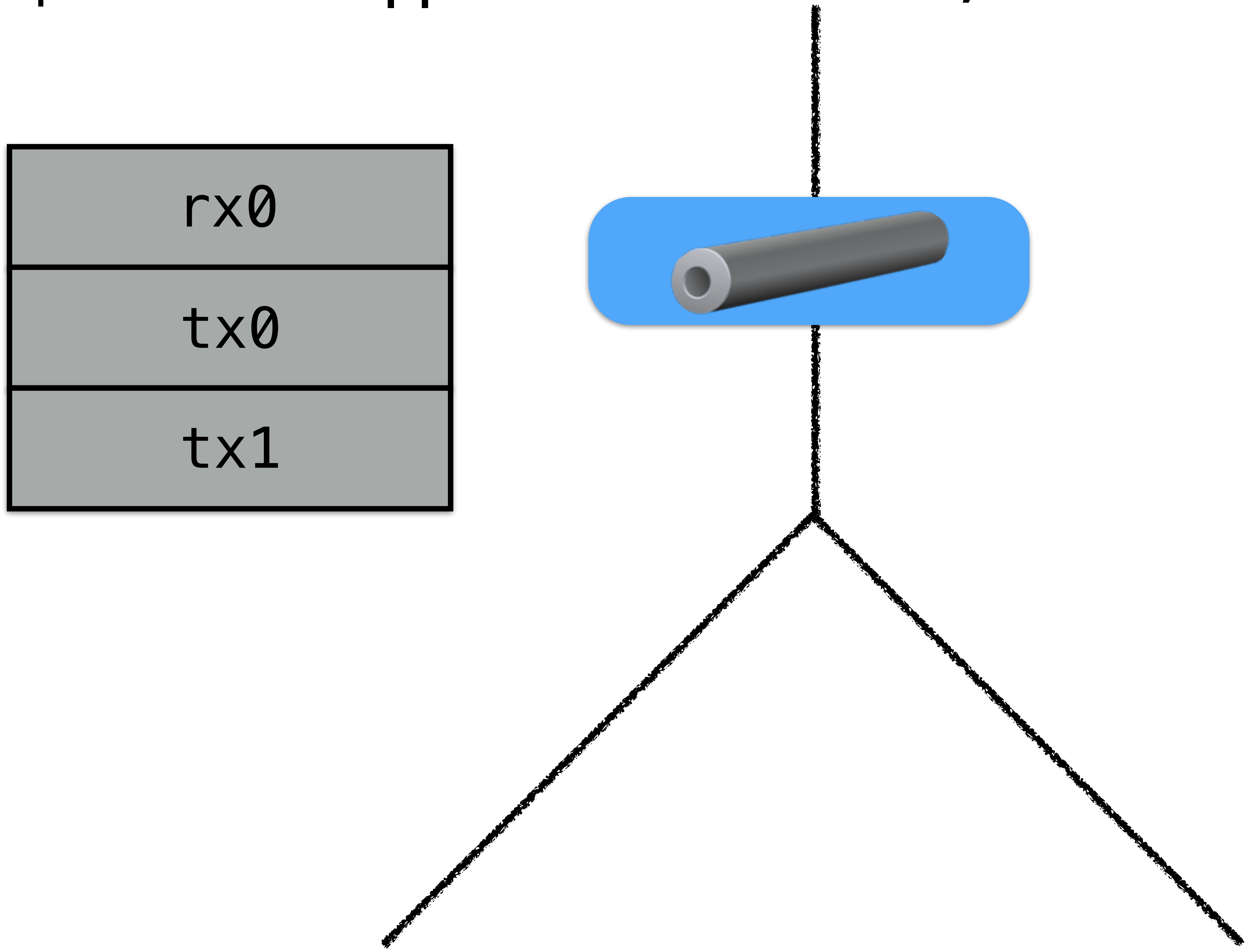
```
let (tx0, rx0) = channel();
let tx1 = tx0.clone();
spawn(move || /* omitted */);
let tx2 = tx0.clone();
spawn(move || /* omitted */);
```



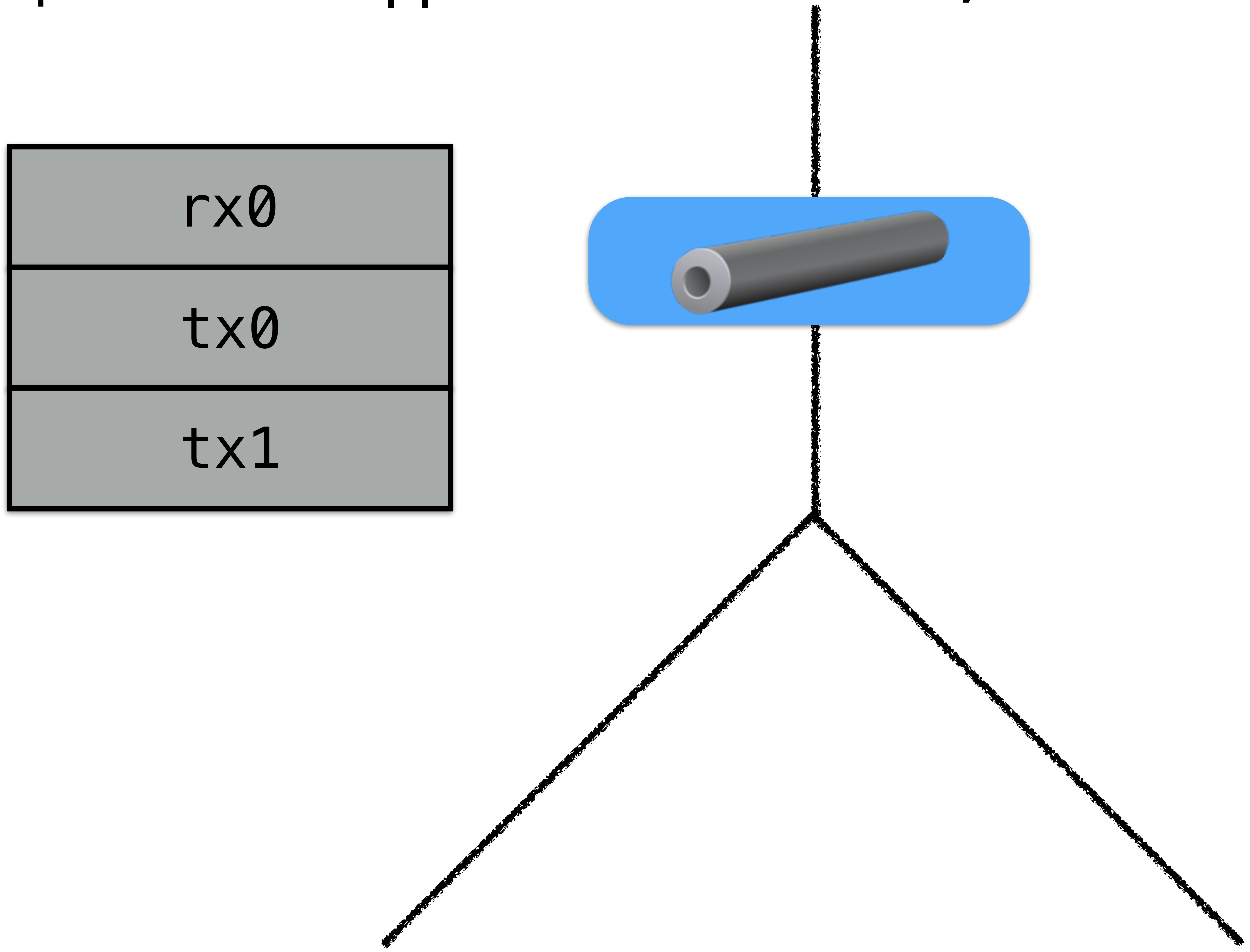
```
let (tx0, rx0) = channel();
let tx1 = tx0.clone();
spawn(move || /* omitted */);
let tx2 = tx0.clone();
spawn(move || /* omitted */);
```



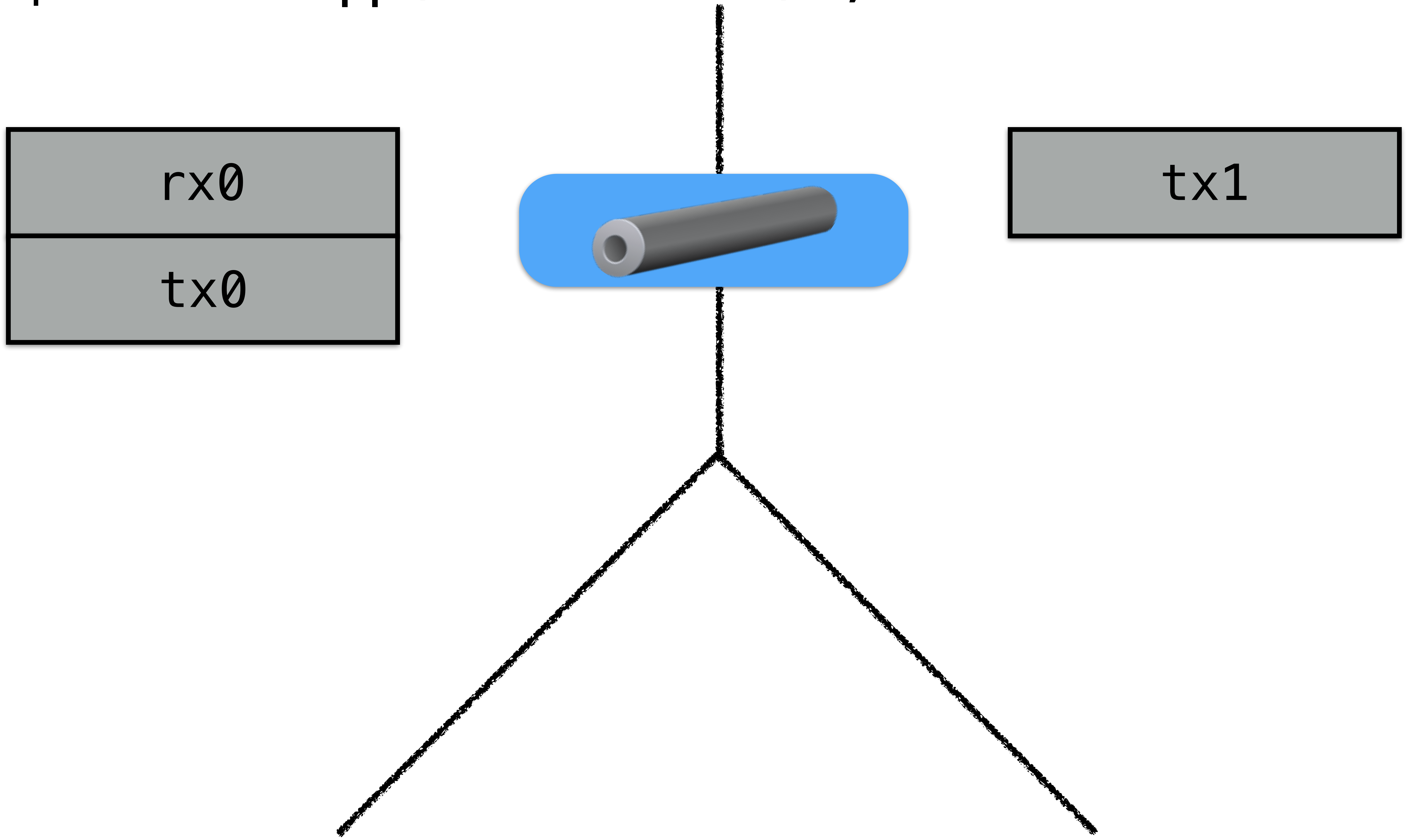
```rust
let (tx0, rx0) = channel();
let tx1 = tx0.clone();
spawn(move || /* omitted */);
let tx2 = tx0.clone();
spawn(move || /* omitted */);
```



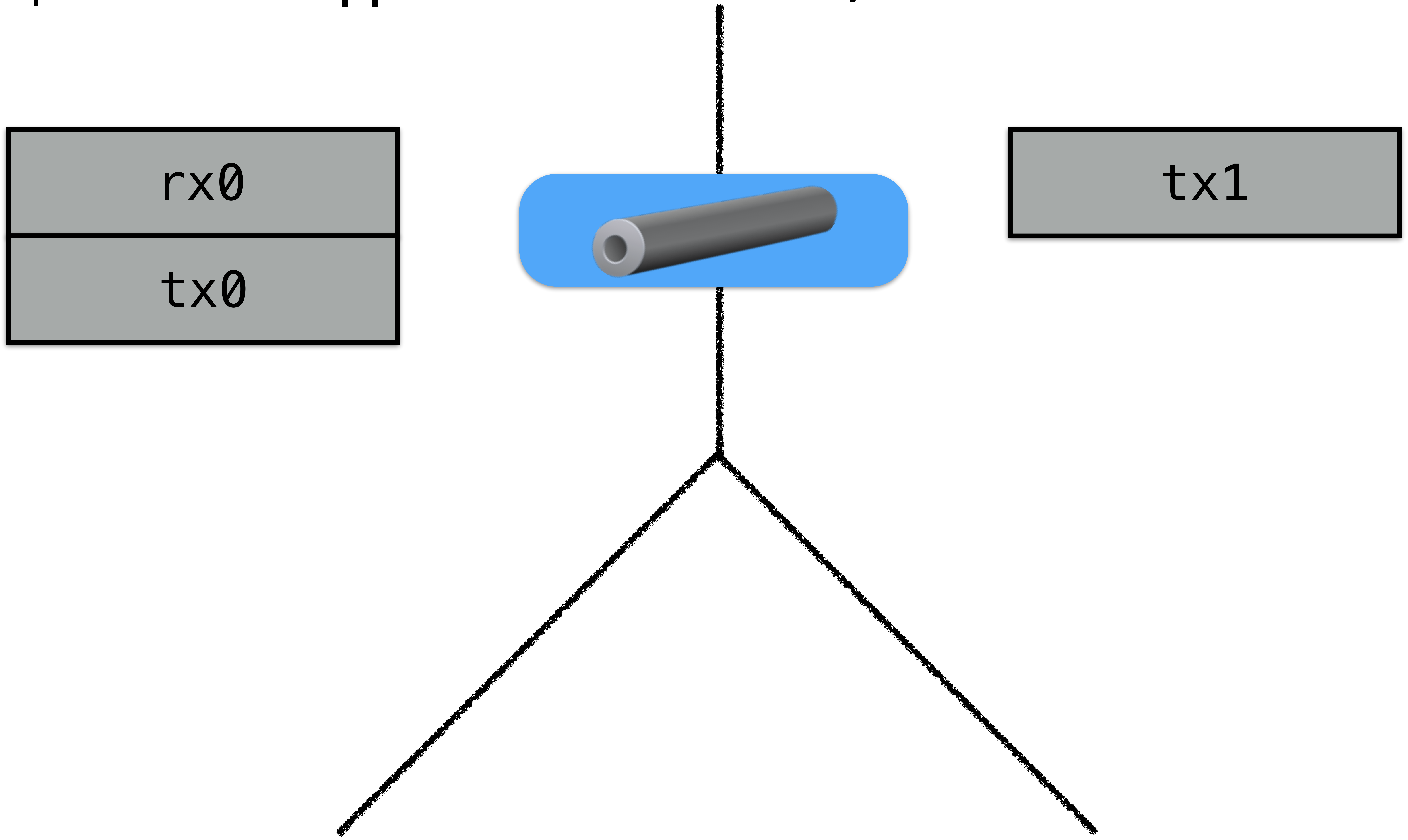
```
let (tx0, rx0) = channel();
let tx1 = tx0.clone();
spawn(move || /* omitted */);
let tx2 = tx0.clone();
spawn(move || /* omitted */);
```



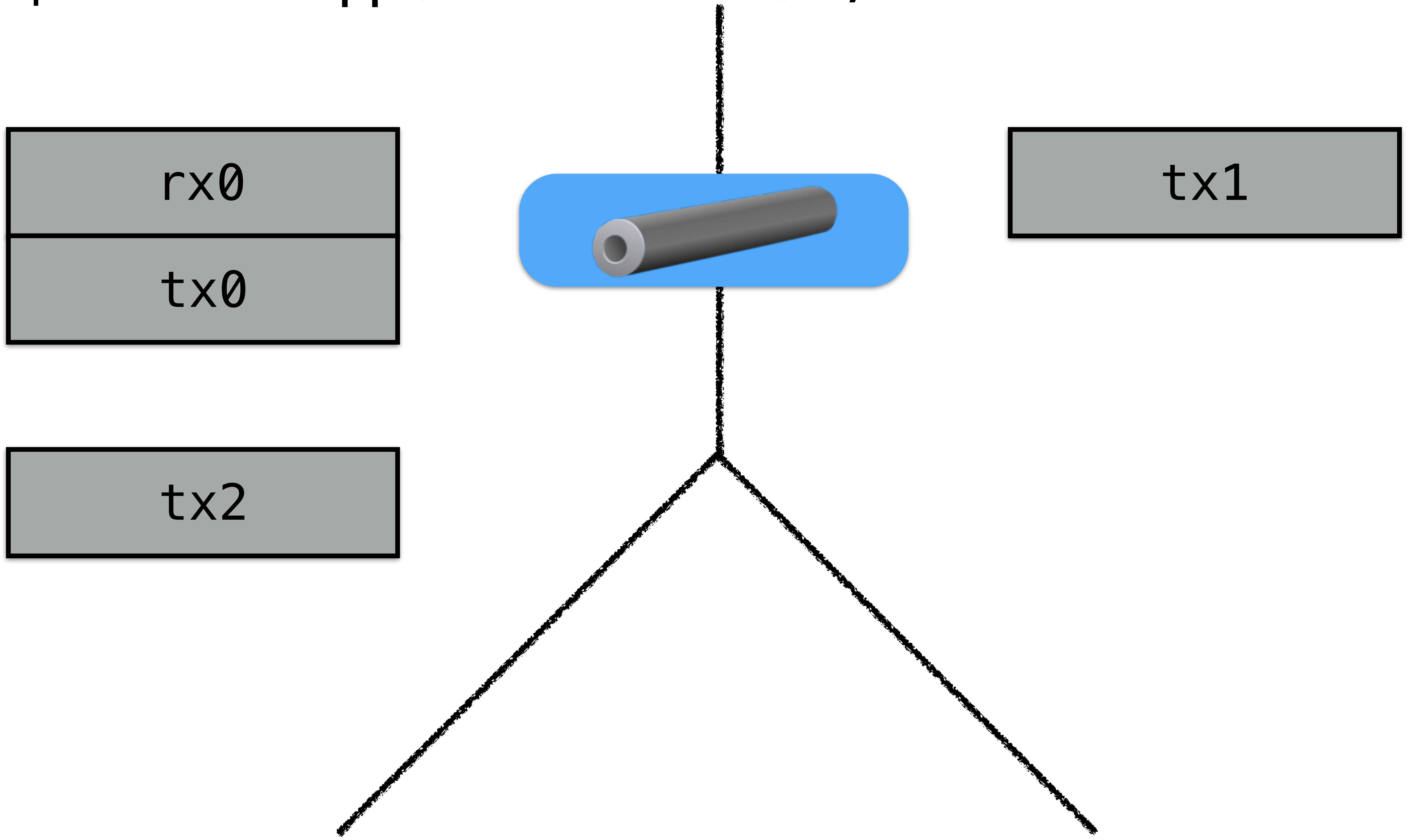
```
let (tx0, rx0) = channel();
let tx1 = tx0.clone();
spawn(move || /* omitted */);
let tx2 = tx0.clone();
spawn(move || /* omitted */);
```



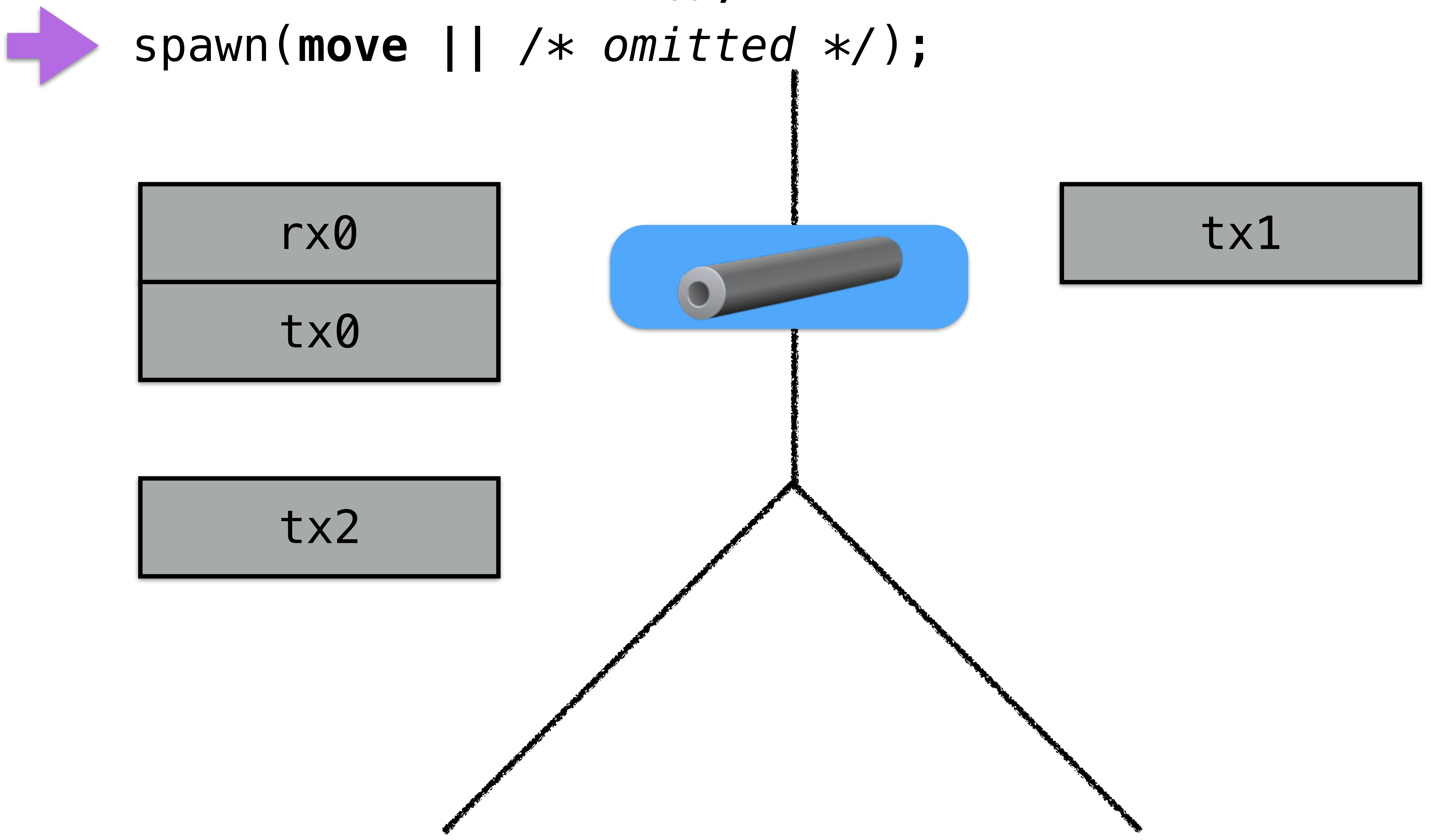
```
let (tx0, rx0) = channel();
let tx1 = tx0.clone();
spawn(move || /* omitted */);
let tx2 = tx0.clone();
spawn(move || /* omitted */);
```



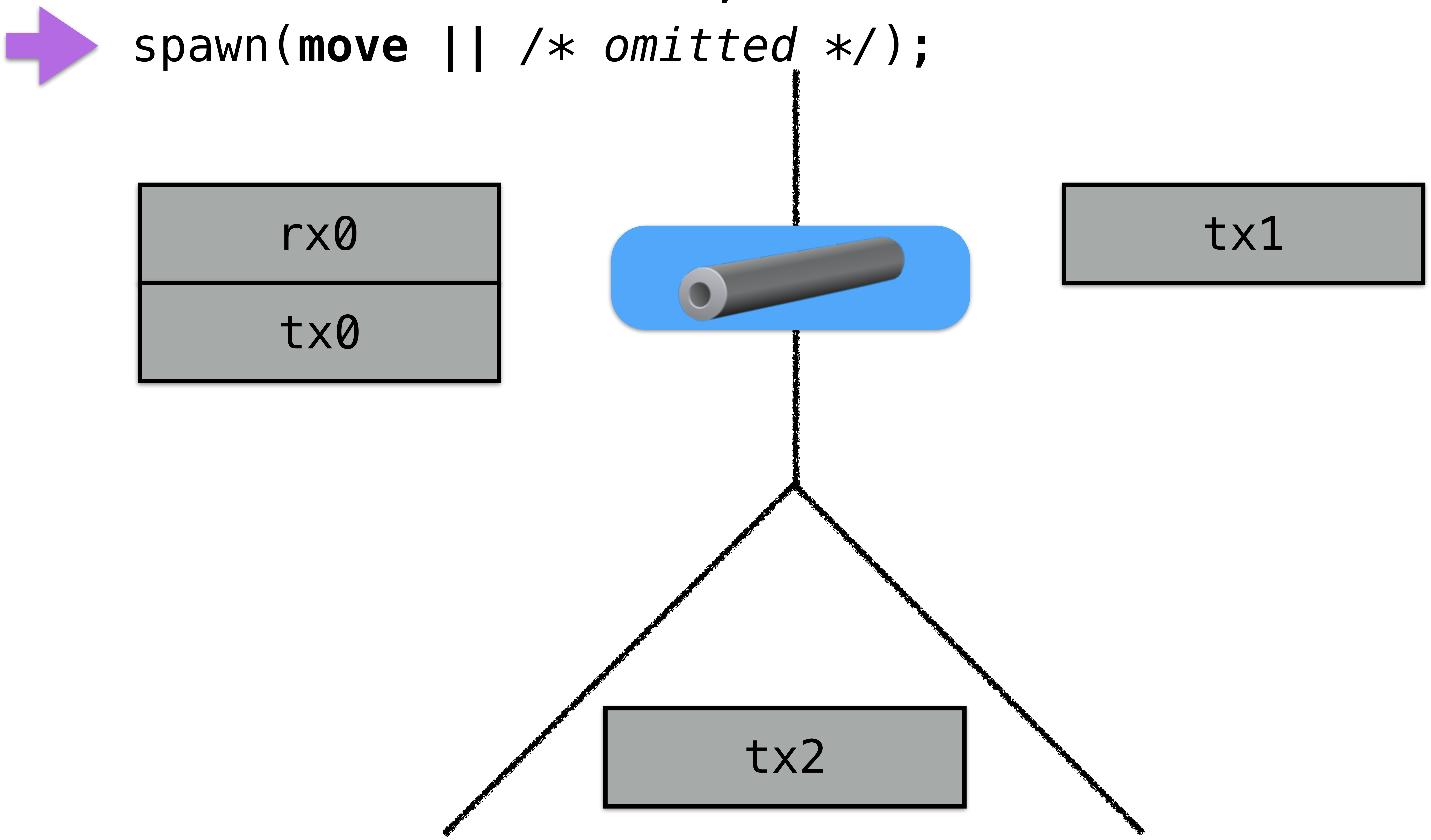
```
let (tx0, rx0) = channel();
let tx1 = tx0.clone();
spawn(move || /* omitted */);
let tx2 = tx0.clone();
spawn(move || /* omitted */);
```



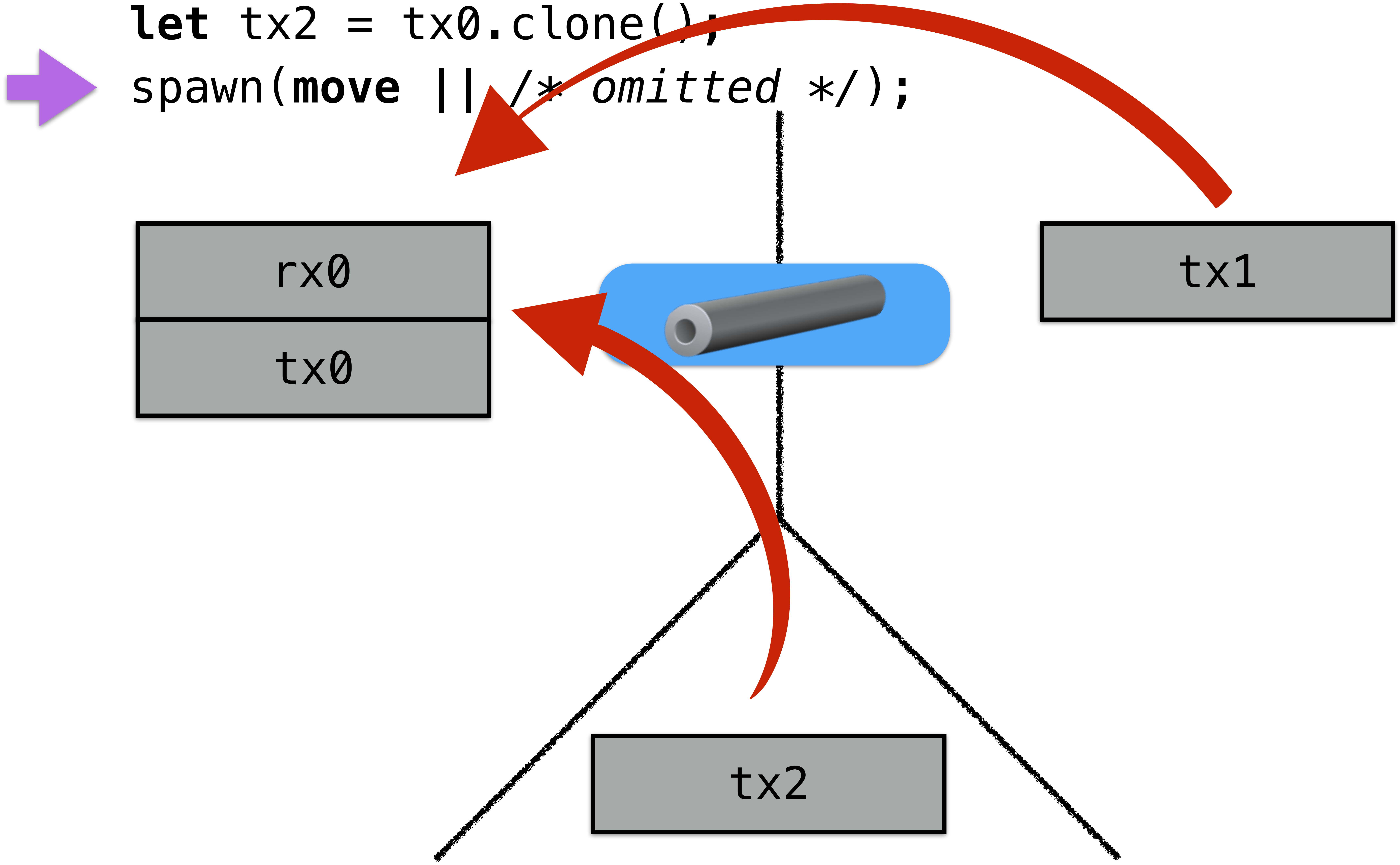
```
let (tx0, rx0) = channel();
let tx1 = tx0.clone();
spawn(move || /* omitted */);
let tx2 = tx0.clone();
spawn(move || /* omitted */);
```



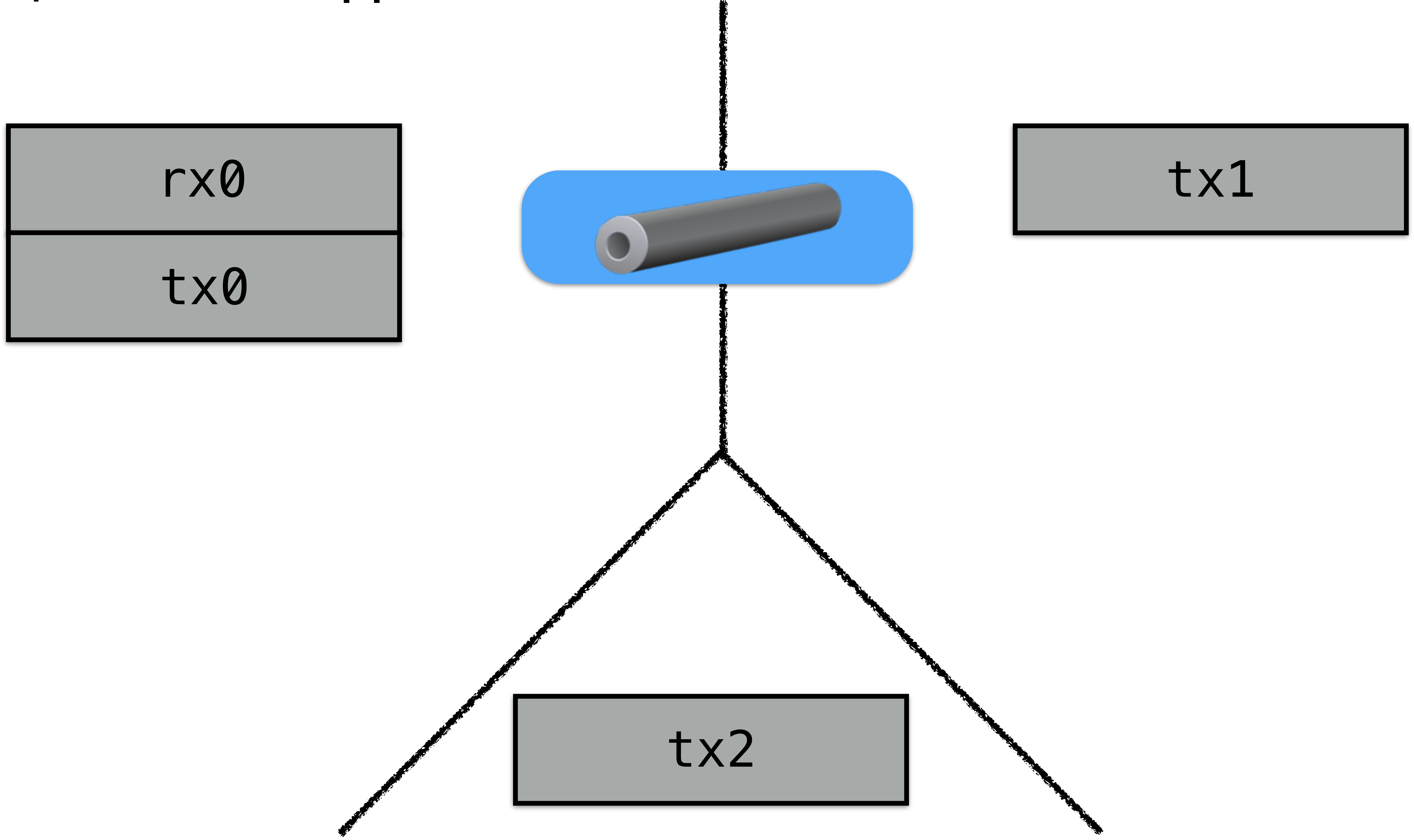
```
let (tx0, rx0) = channel();
let tx1 = tx0.clone();
spawn(move || /* omitted */);
let tx2 = tx0.clone();
spawn(move || /* omitted */);
```



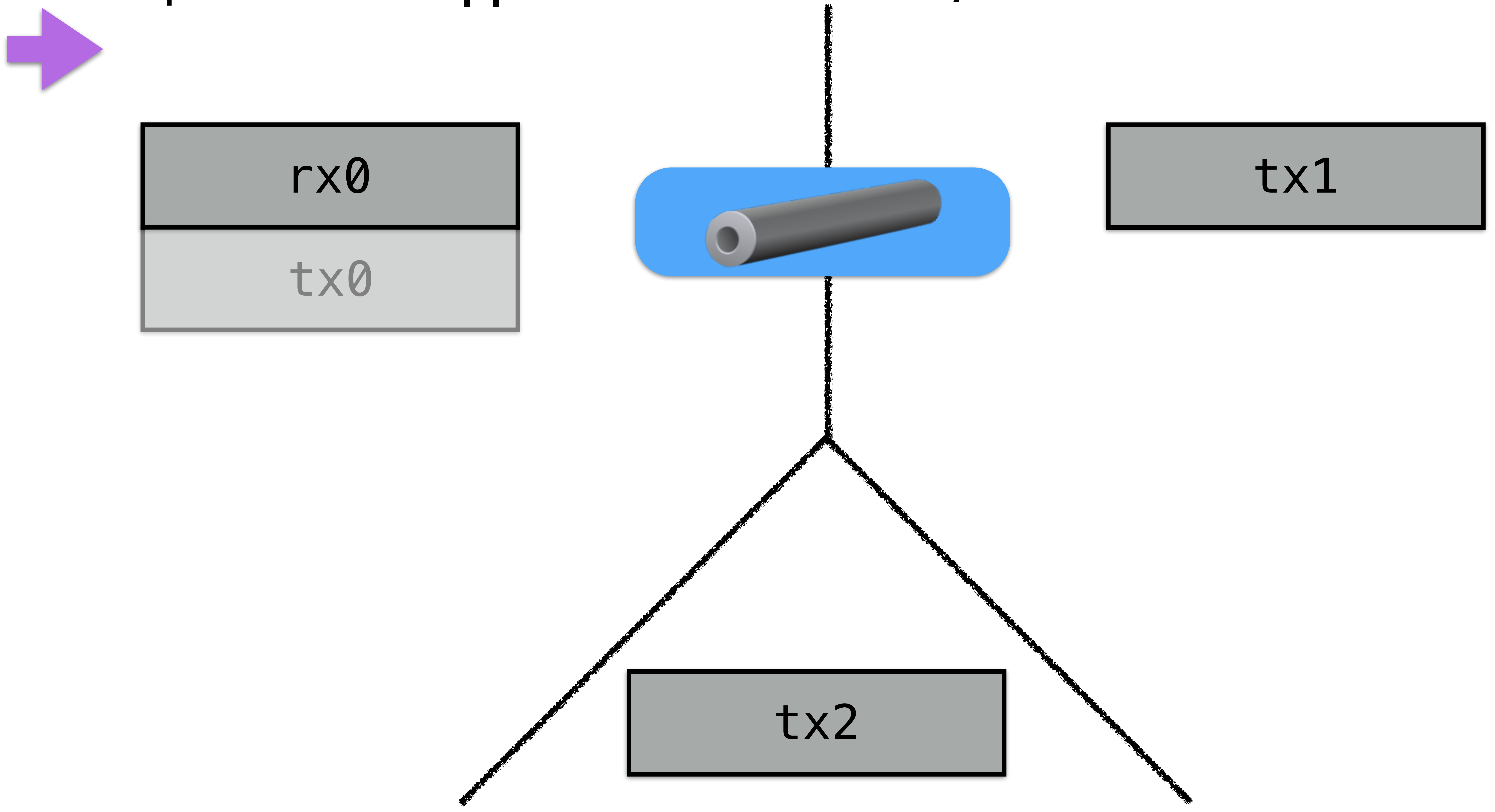
```
let (tx0, rx0) = channel();
let tx1 = tx0.clone();
spawn(move || /* omitted */);
let tx2 = tx0.clone();
spawn(move || /* omitted */);
```



```
let (tx0, rx0) = channel();
let tx1 = tx0.clone();
spawn(move || /* omitted */);
let tx2 = tx0.clone();
spawn(move || /* omitted */);
```



```
let (tx0, rx0) = channel();
let tx1 = tx0.clone();
spawn(move || /* omitted */);
let tx2 = tx0.clone();
spawn(move || /* omitted */);
```



Locks

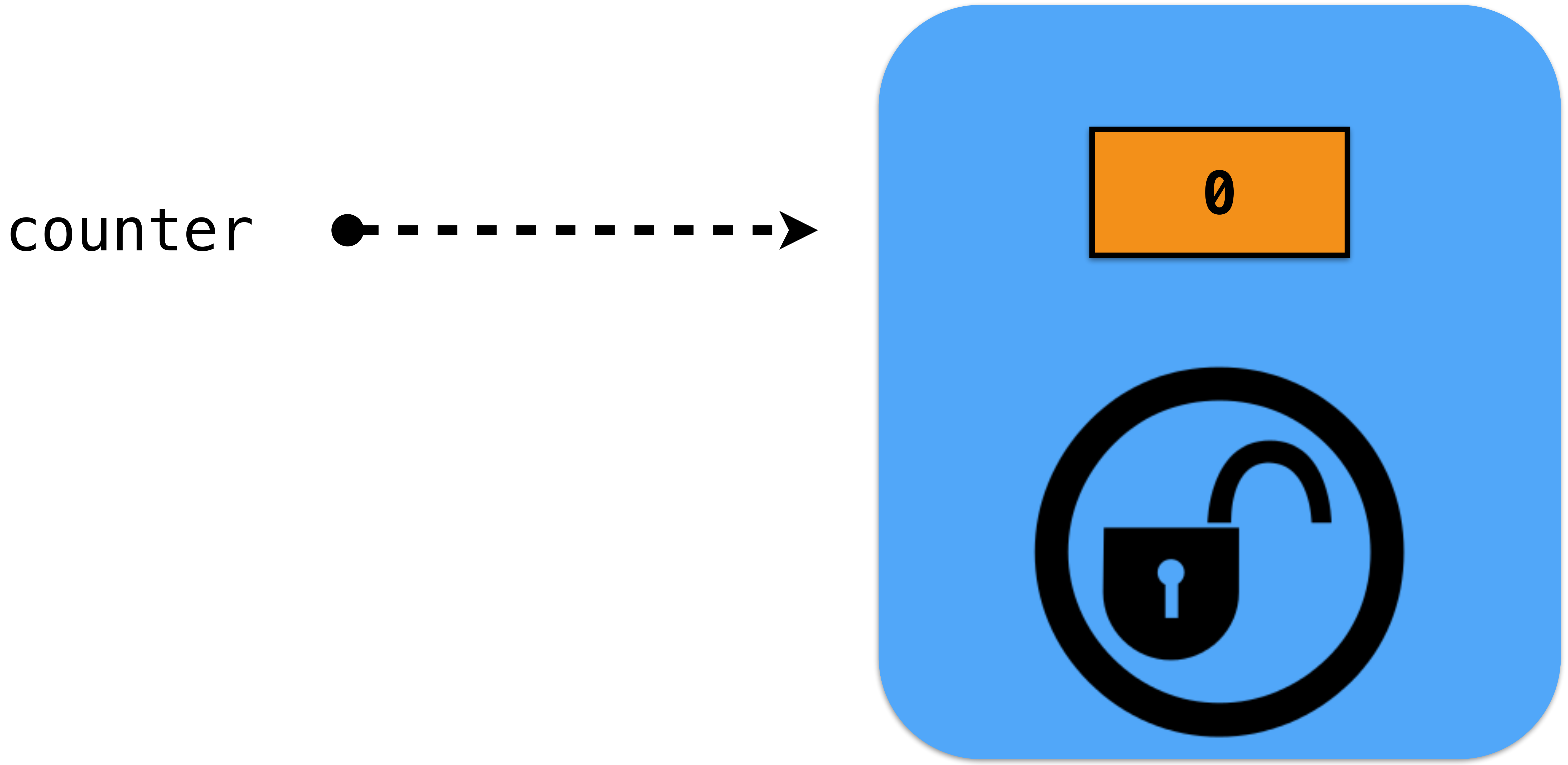
```rust
fn sync_inc(counter: &Arc<Mutex<i32>>) {
  let mut data = counter.lock().unwrap();
  *data += 1;
}
```

https://commons.wikimedia.org/wiki/File:No-DRM_lock.svg

```
fn sync_inc(counter: &Arc<Mutex<i32>>) {
  let mut data = counter.lock().unwrap();
  *data += 1;
}
```

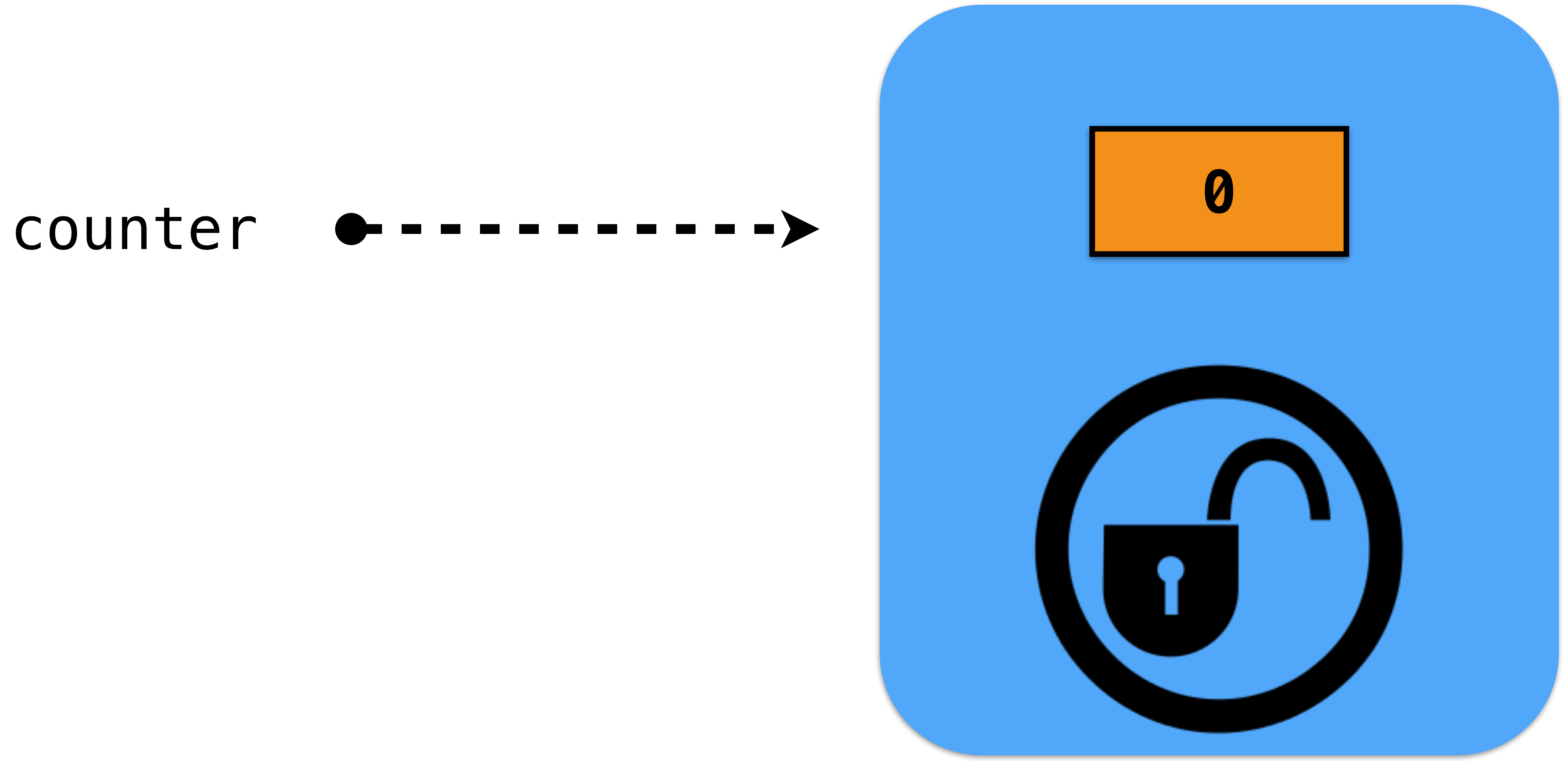
https://commons.wikimedia.org/wiki/File:No-DRM_lock.svg

```
fn sync_inc(counter: &Arc<Mutex<i32>>) {
  let mut data = counter.lock().unwrap();
  *data += 1;
}
```



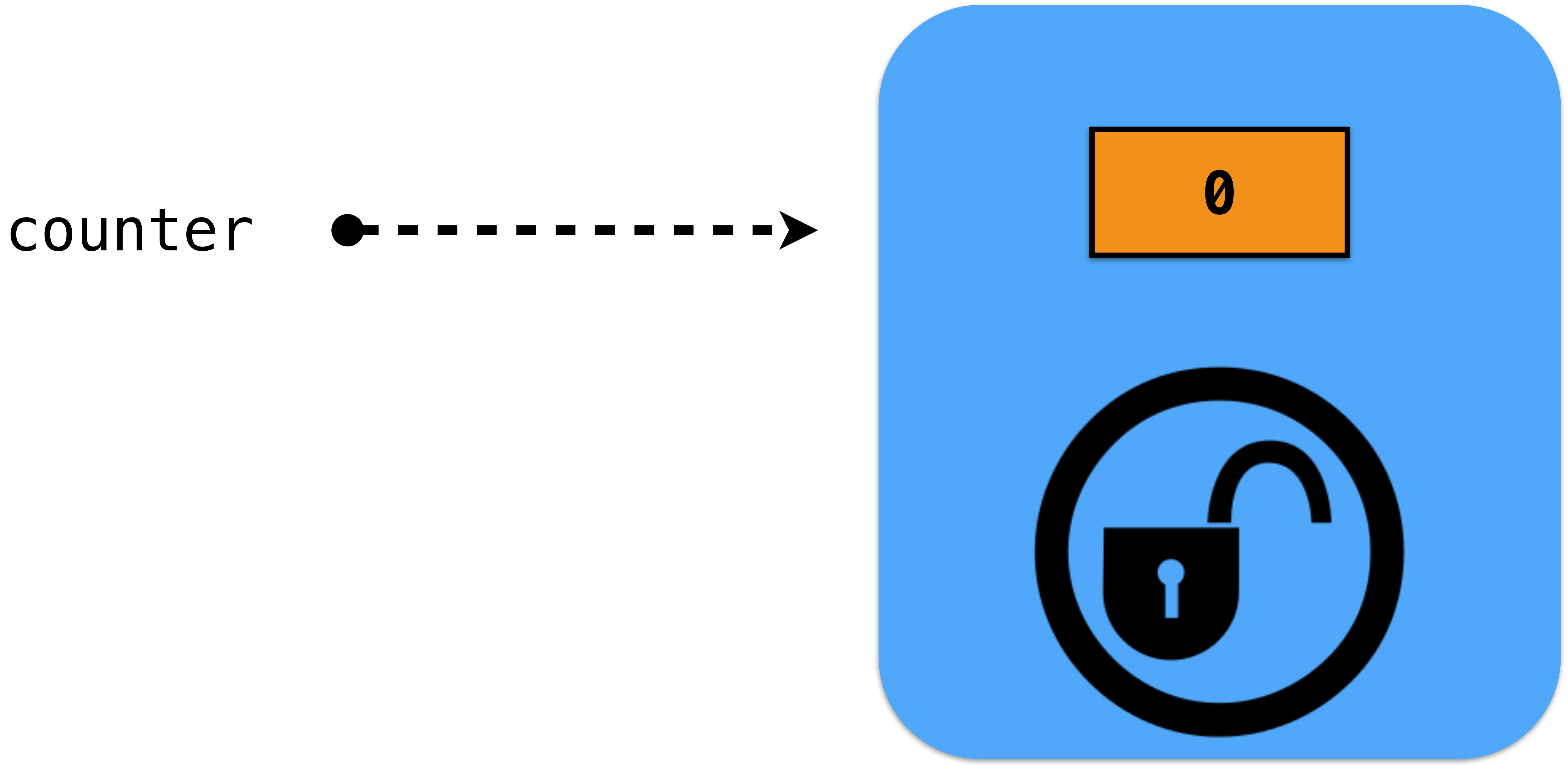
https://commons.wikimedia.org/wiki/File:No-DRM_lock.svg

```
fn sync_inc(counter: &Arc<Mutex<i32>>) {
  let mut data = counter.lock().unwrap();
  *data += 1;
}
```



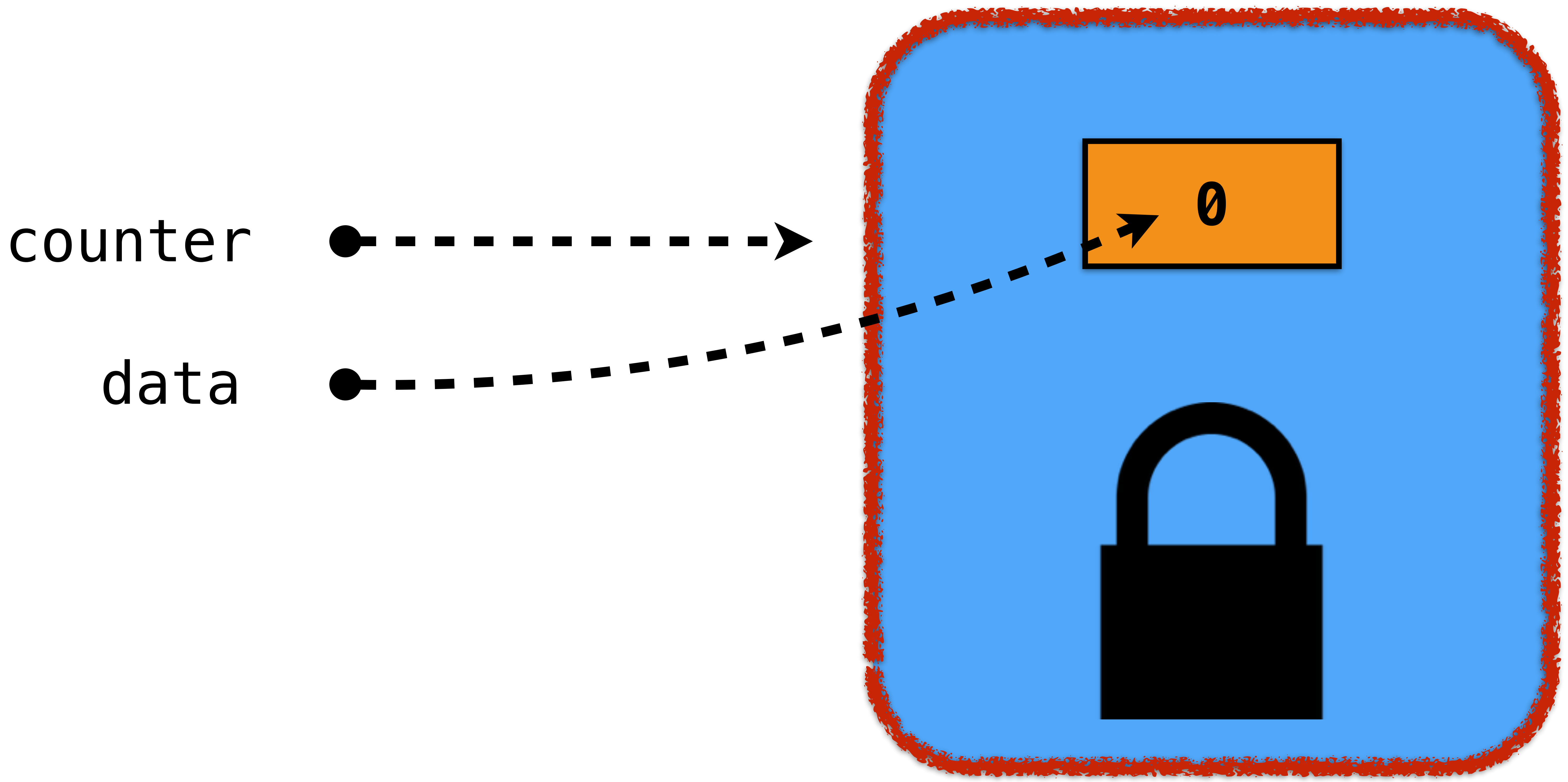
https://commons.wikimedia.org/wiki/File:No-DRM_lock.svg

```rust
fn sync_inc(counter: &Arc<Mutex<i32>>) {
  let mut data = counter.lock().unwrap();
  *data += 1;
}
```



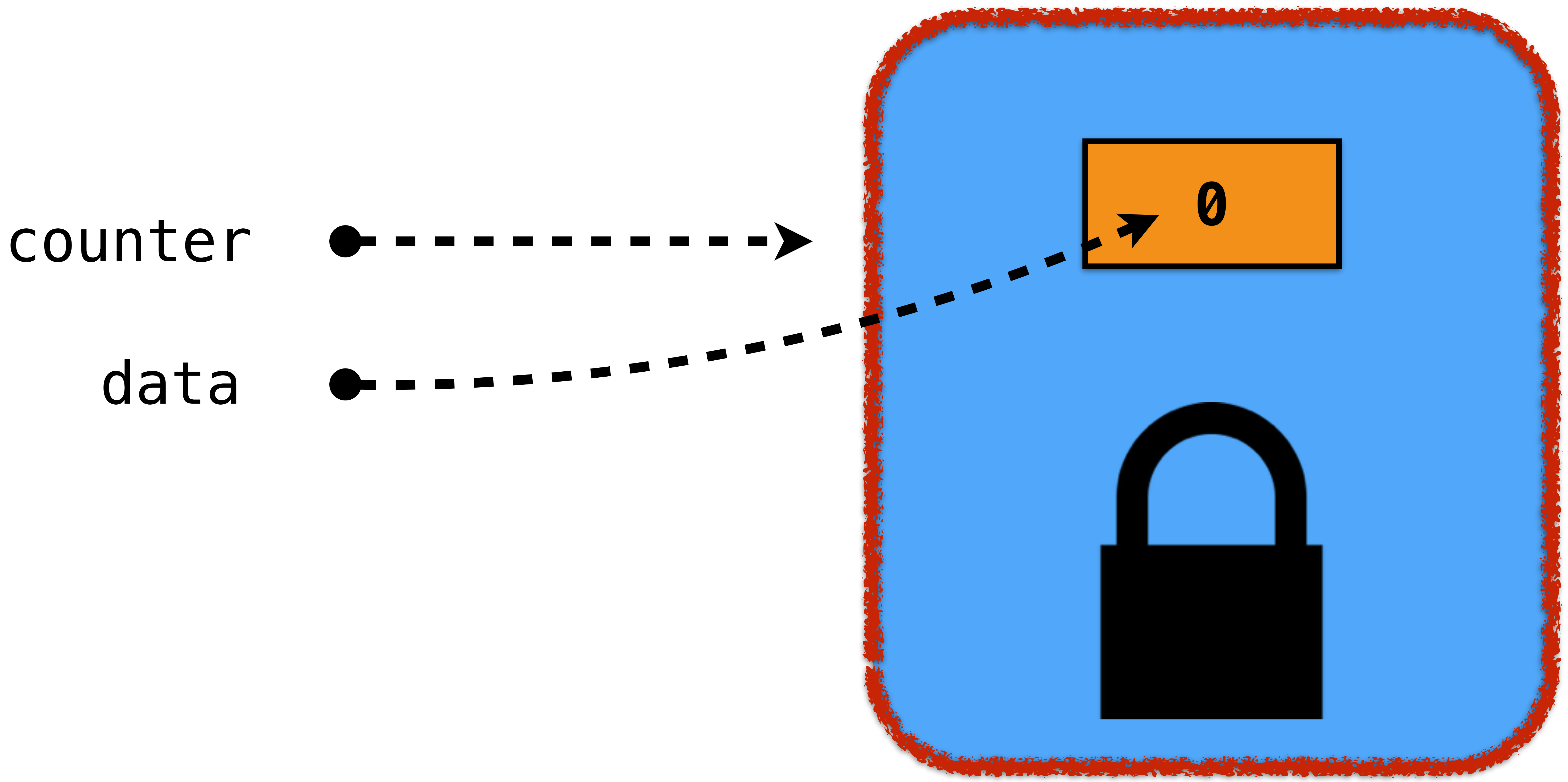
https://commons.wikimedia.org/wiki/File:No-DRM_lock.svg

```
fn sync_inc(counter: &Arc<Mutex<i32>>) {
  let mut data = counter.lock().unwrap();
  *data += 1;
}
```



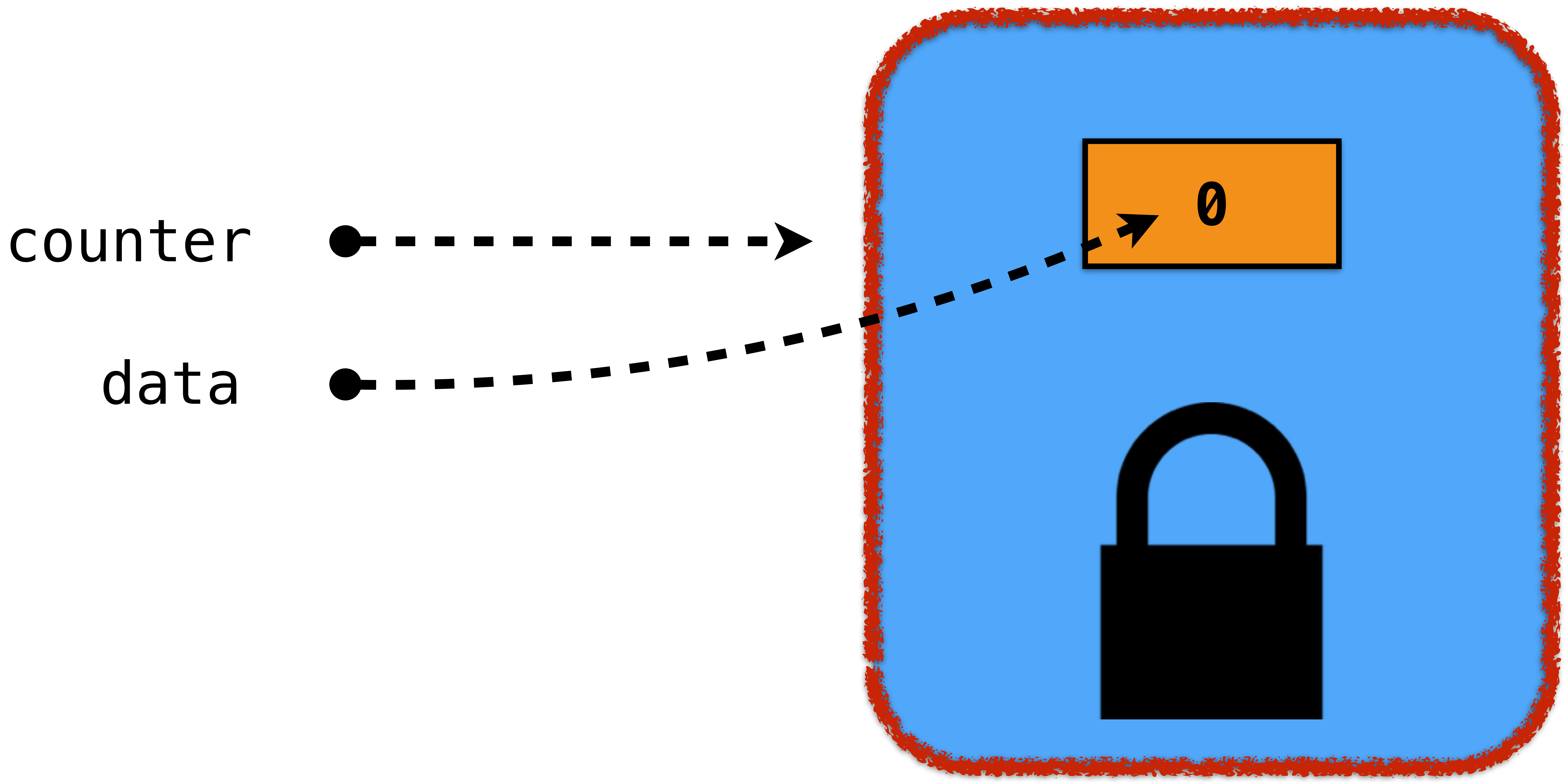
https://commons.wikimedia.org/wiki/File:No-DRM_lock.svg

```
fn sync_inc(counter: &Arc<Mutex<i32>>) {
  let mut data = counter.lock().unwrap();
  *data += 1;
}
```



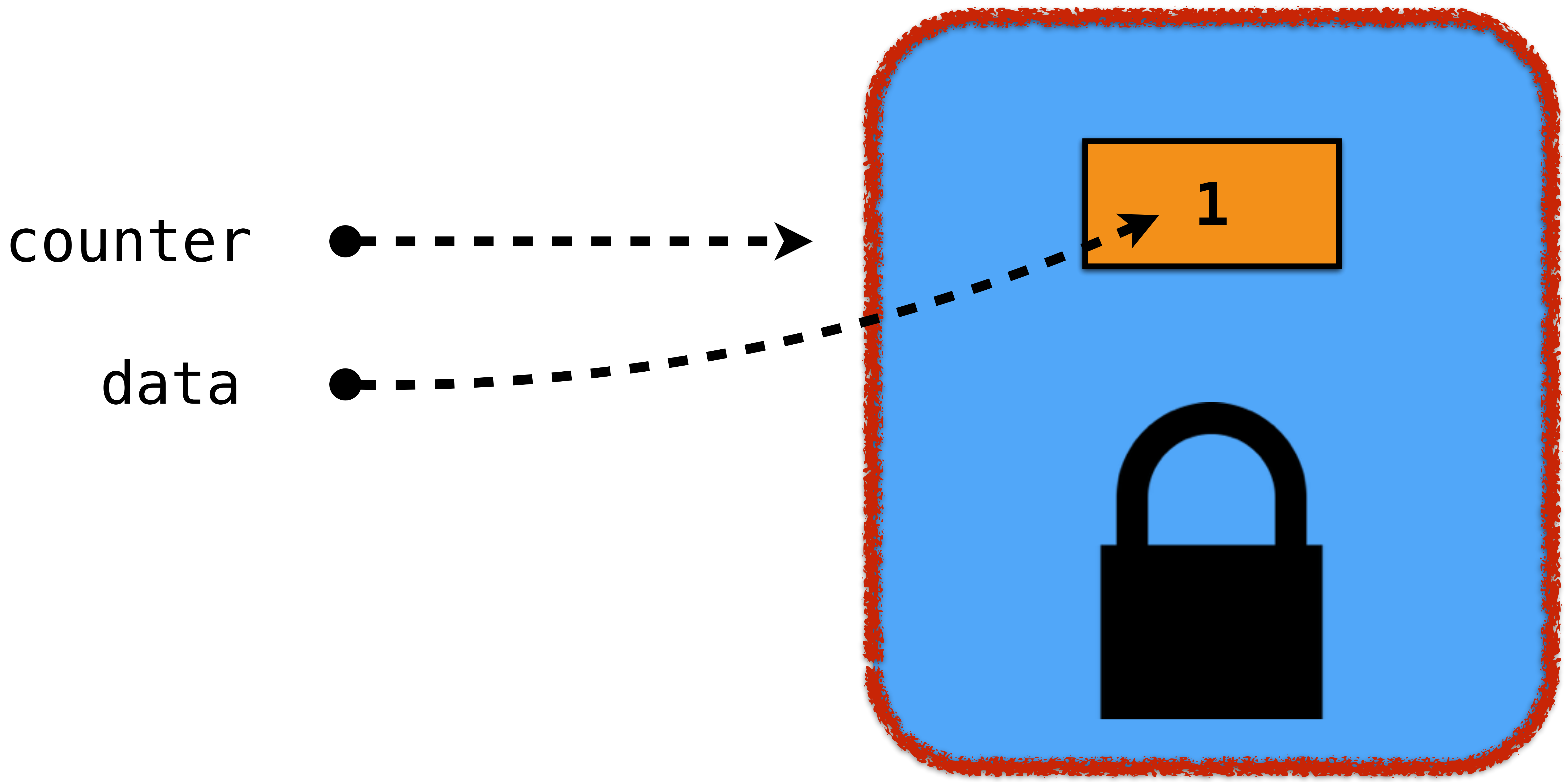
https://commons.wikimedia.org/wiki/File:No-DRM_lock.svg

```rust
fn sync_inc(counter: &Arc<Mutex<i32>>) {
  let mut data = counter.lock().unwrap();
  *data += 1;
}
```



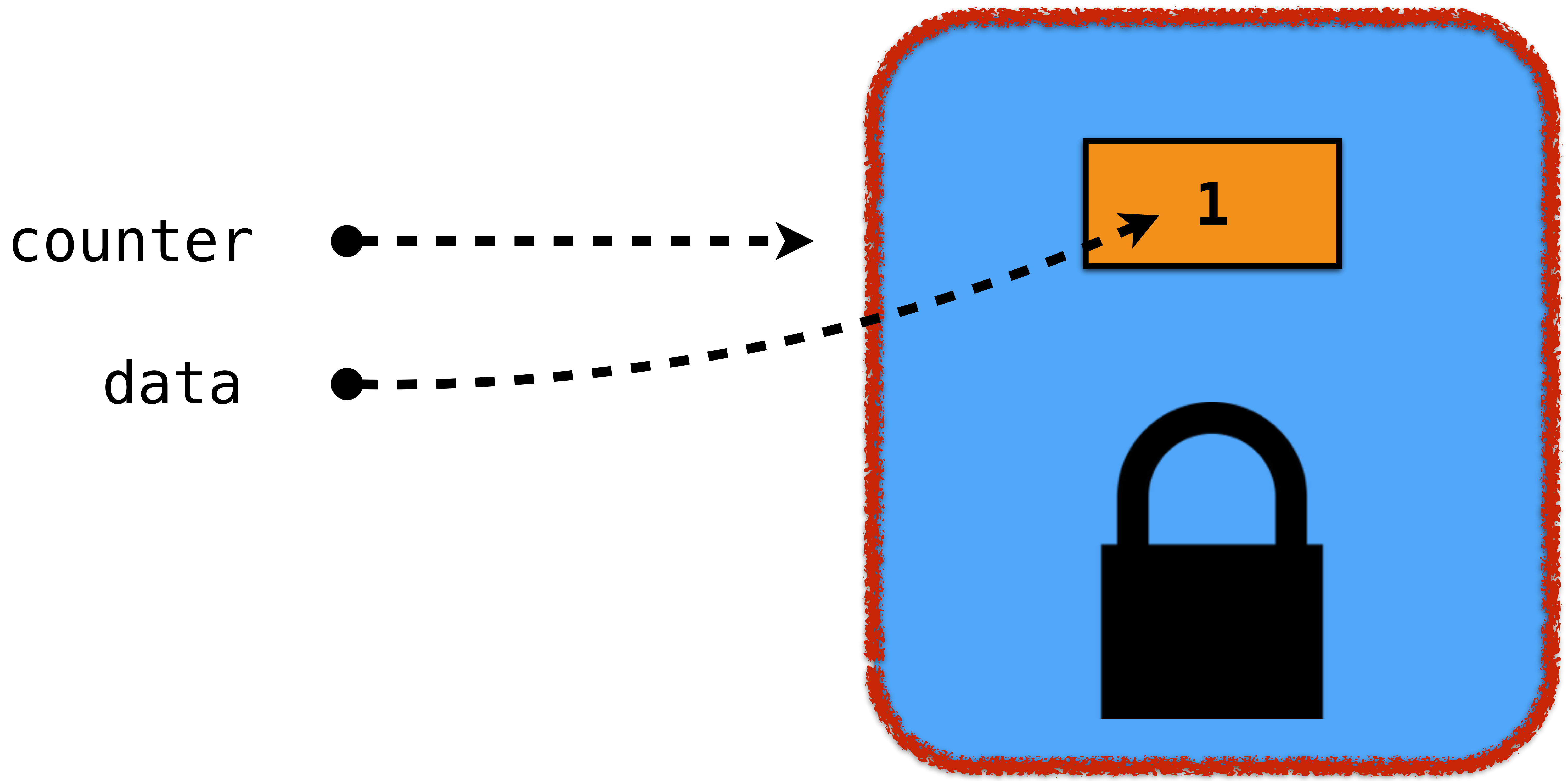
https://commons.wikimedia.org/wiki/File:No-DRM_lock.svg

```
fn sync_inc(counter: &Arc<Mutex<i32>>) {
  let mut data = counter.lock().unwrap();
  *data += 1;
}
```



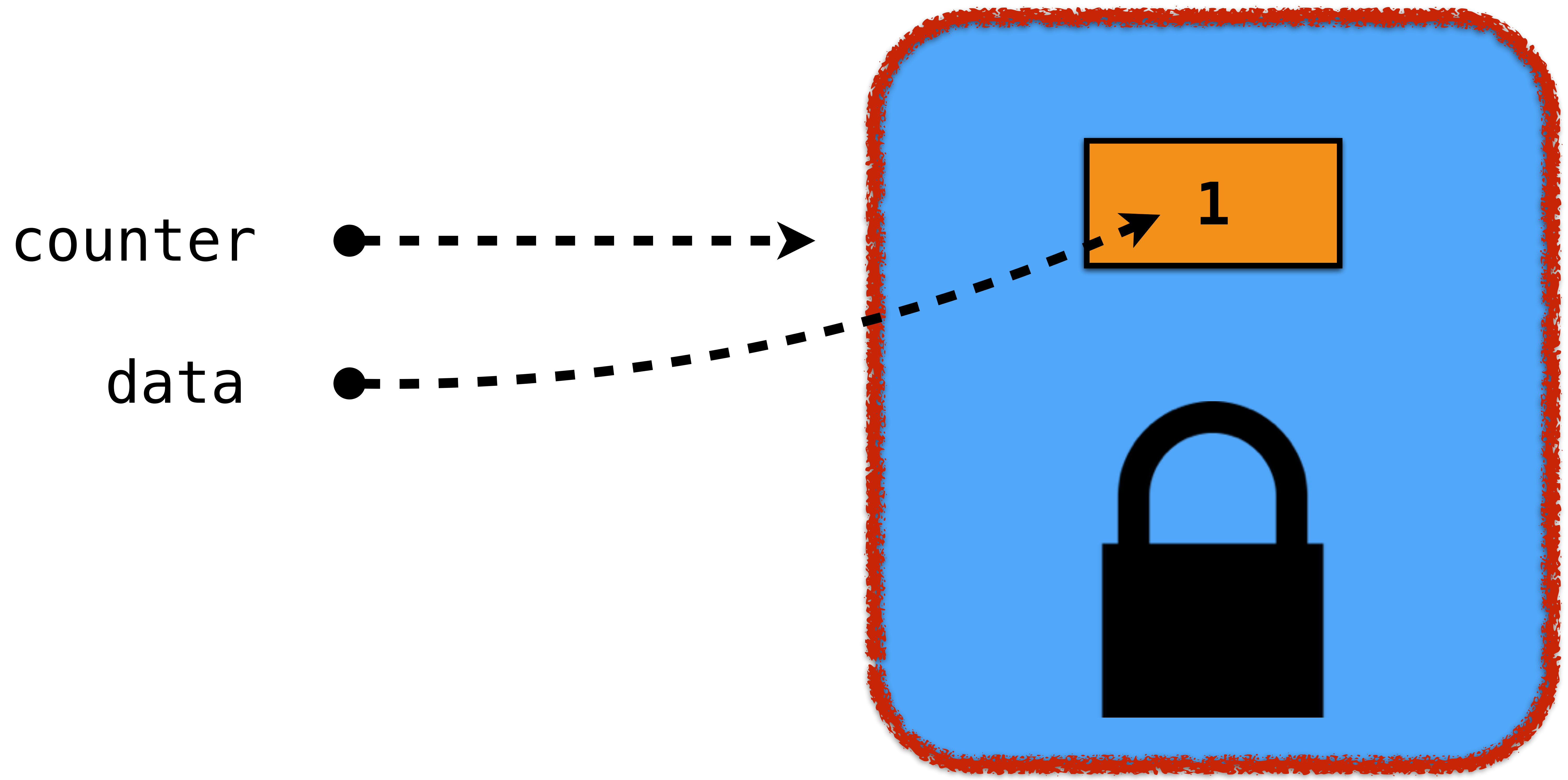
https://commons.wikimedia.org/wiki/File:No-DRM_lock.svg

```
fn sync_inc(counter: &Arc<Mutex<i32>>) {
  let mut data = counter.lock().unwrap();
  *data += 1;
}
```



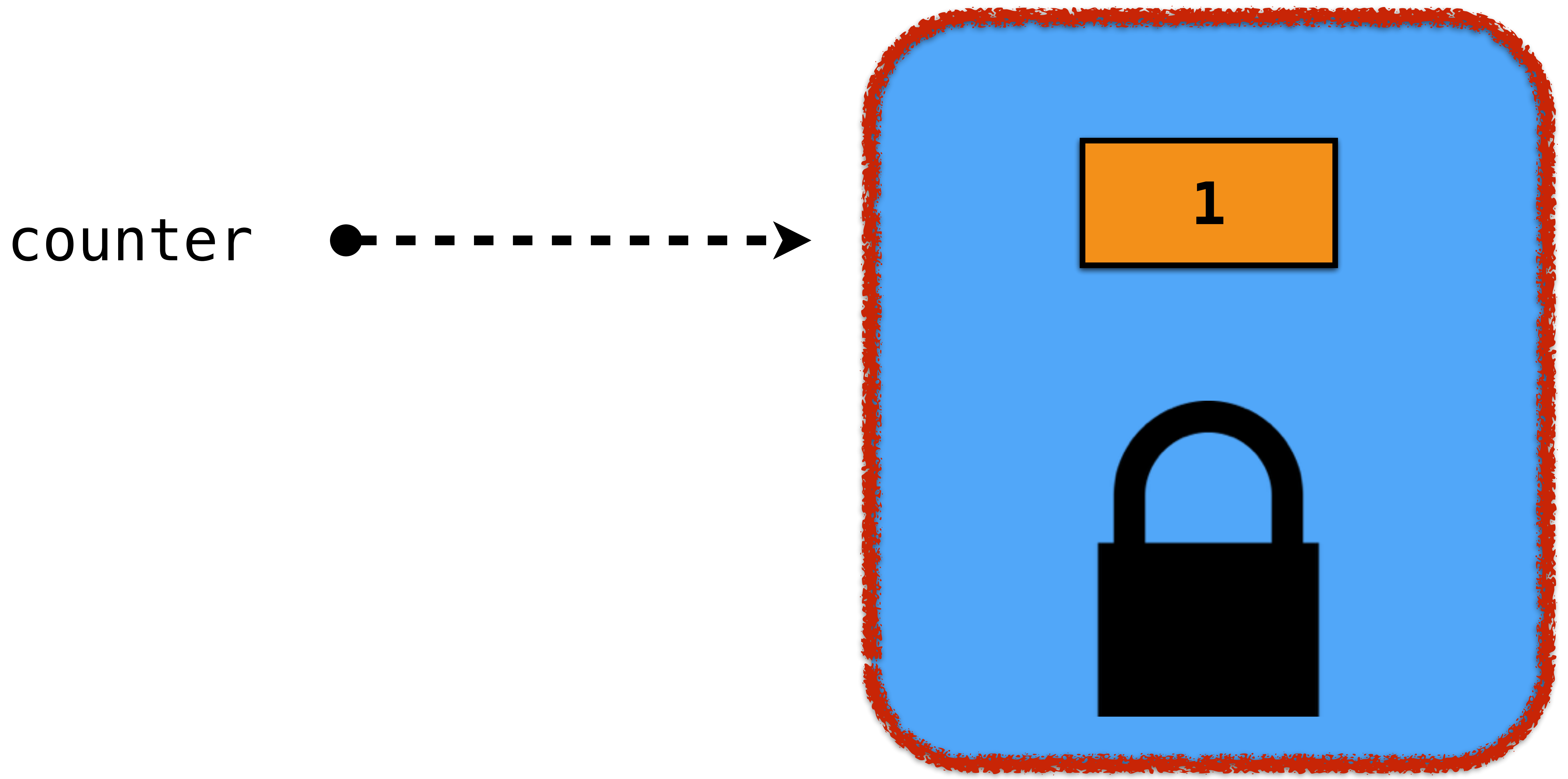
https://commons.wikimedia.org/wiki/File:No-DRM_lock.svg

```
fn sync_inc(counter: &Arc<Mutex<i32>>) {
  let mut data = counter.lock().unwrap();
  *data += 1;
}
```



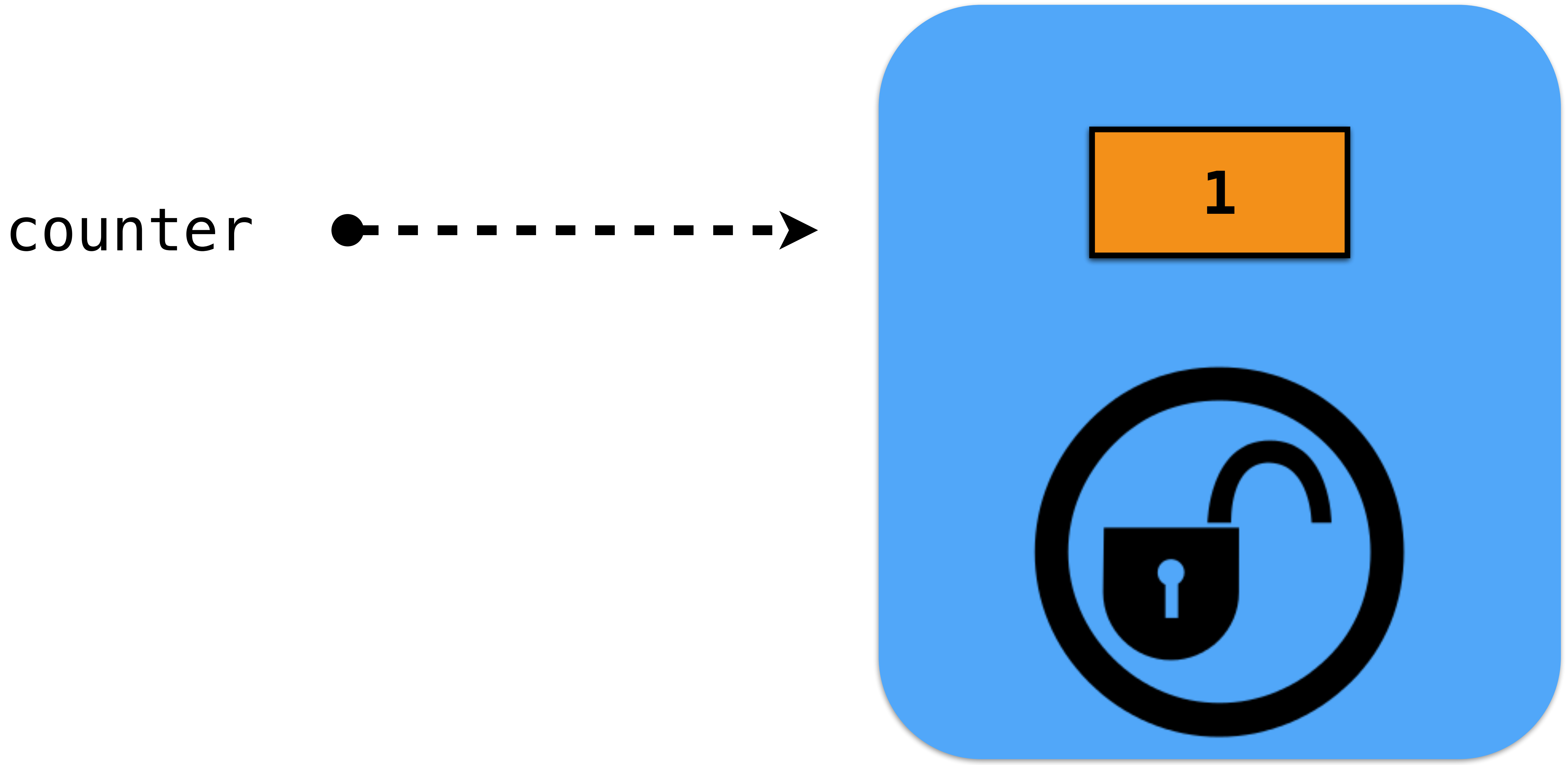
https://commons.wikimedia.org/wiki/File:No-DRM_lock.svg

```rust
fn sync_inc(counter: &Arc<Mutex<i32>>) {
  let mut data = counter.lock().unwrap();
  *data += 1;
}
```



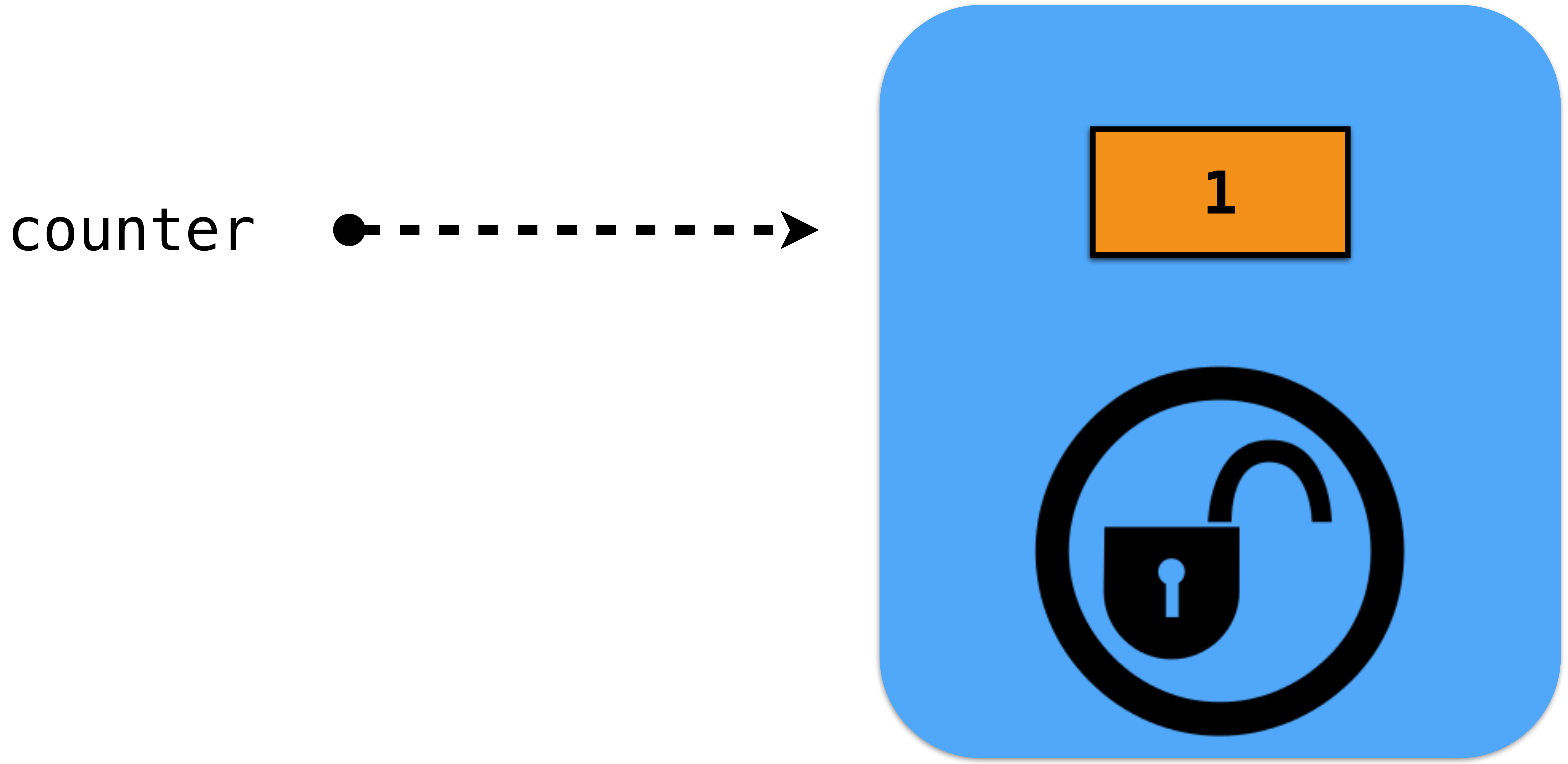
https://commons.wikimedia.org/wiki/File:No-DRM_lock.svg

```
fn sync_inc(counter: &Arc<Mutex<i32>>) {
  let mut data = counter.lock().unwrap();
  *data += 1;
}
```



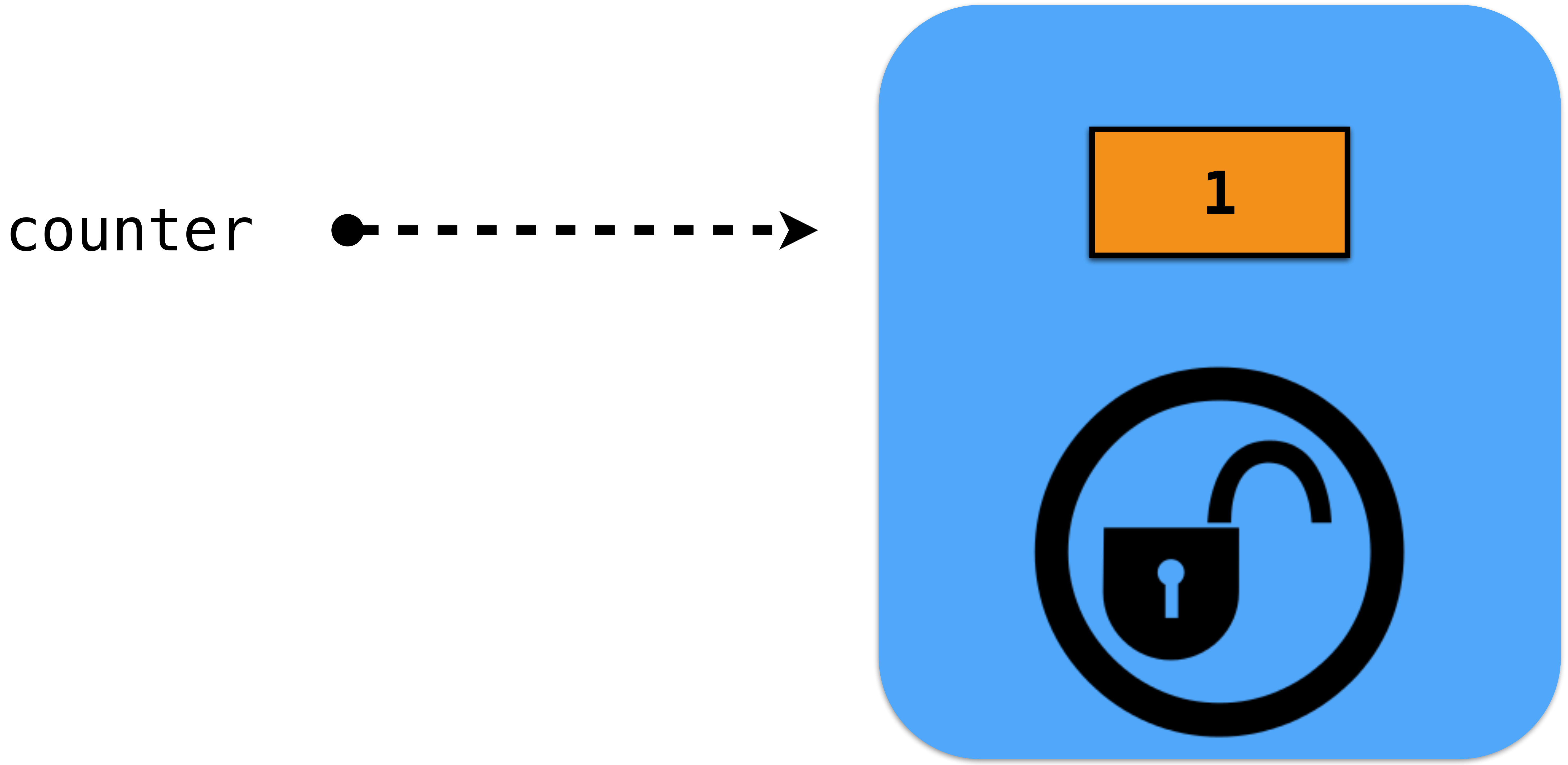
https://commons.wikimedia.org/wiki/File:No-DRM_lock.svg

```
fn sync_inc(counter: &Arc<Mutex<i32>>) {
  let mut data = counter.lock().unwrap();
  *data += 1;
}
```



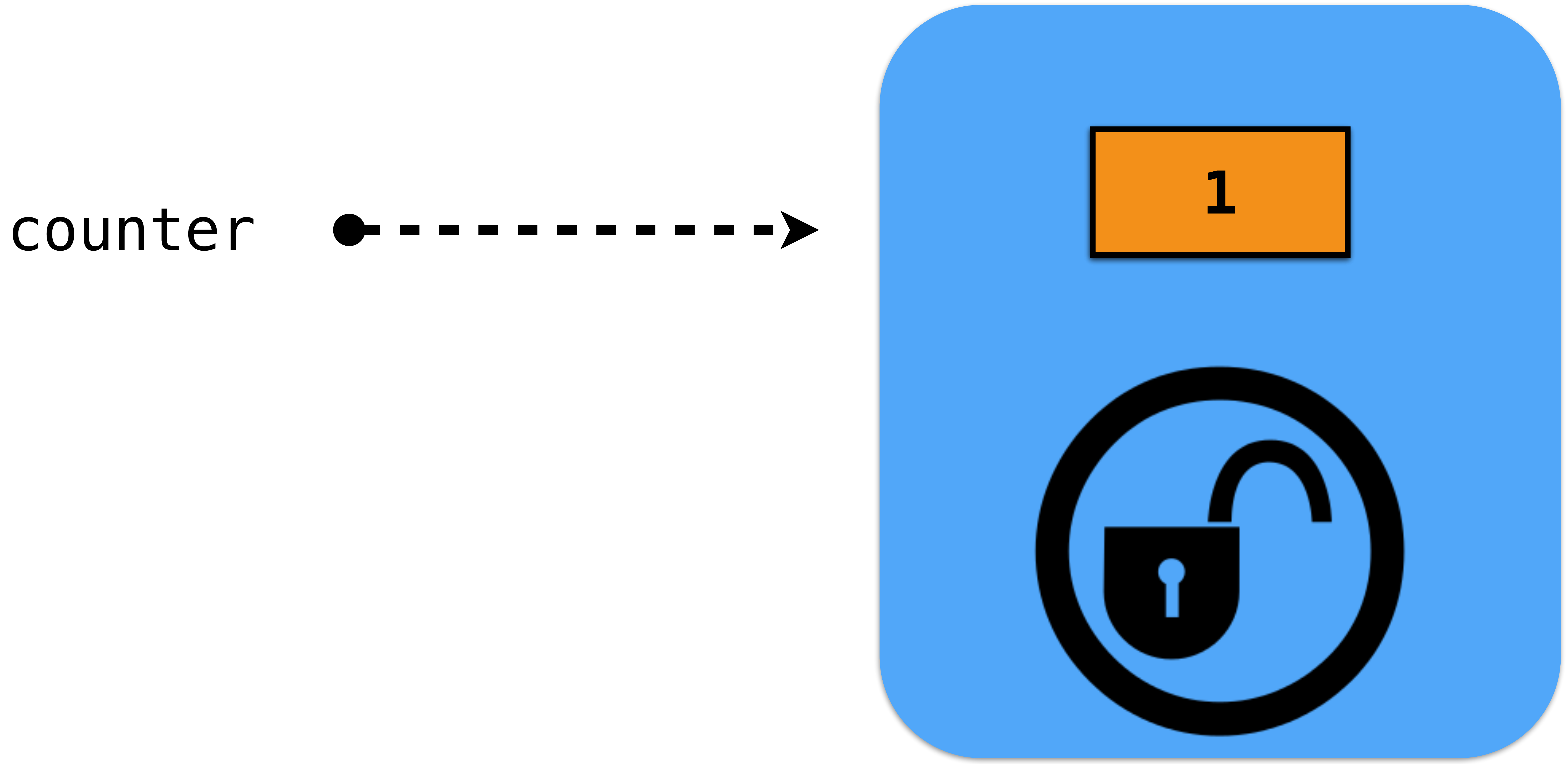
https://commons.wikimedia.org/wiki/File:No-DRM_lock.svg

```
fn sync_inc(counter: &Arc<Mutex<i32>>) {
  let mut data = counter.lock().unwrap();
  *data += 1;
}
```



https://commons.wikimedia.org/wiki/File:No-DRM_lock.svg

```
fn sync_inc(counter: &Arc<Mutex<i32>>) {
  let mut data = counter.lock().unwrap();
  *data += 1;
}
```



https://commons.wikimedia.org/wiki/File:No-DRM_lock.svg

Exercise: **threads**

http://www.rust-tutorials.com/RustConf17/

Cheat sheet:

```
let (name, sum) = thread.join().unwrap();
```

```
use std::sync::mpsc::channel;
let (rx, tx) = channel();
tx.send(value).unwrap();
let value = rx.recv().unwrap();
```

```
use std::sync::{Arc, Mutex};
let mutex = Arc::new(Mutex::new(data));
let data = mutex.lock().unwrap();
```

http://doc.rust-lang.org/std

Static checking for thread safety

```
fn send<T: Send>(&self, t: T)
```

Only “sendable” types

Static checking for thread safety

```
fn send<T: Send>(&self, t: T)
```

Only “sendable” types

```
Arc<Vec<int>>: Send
Rc<Vec<int>> : !Send
```

Thanks for listening!