CS3211 Tutorial 7

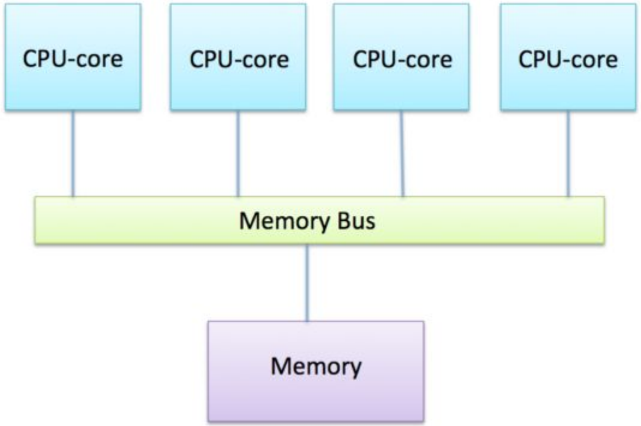
Classic concurrency problems in C++ and Go

Simon
Adapted from Sriram's Slides

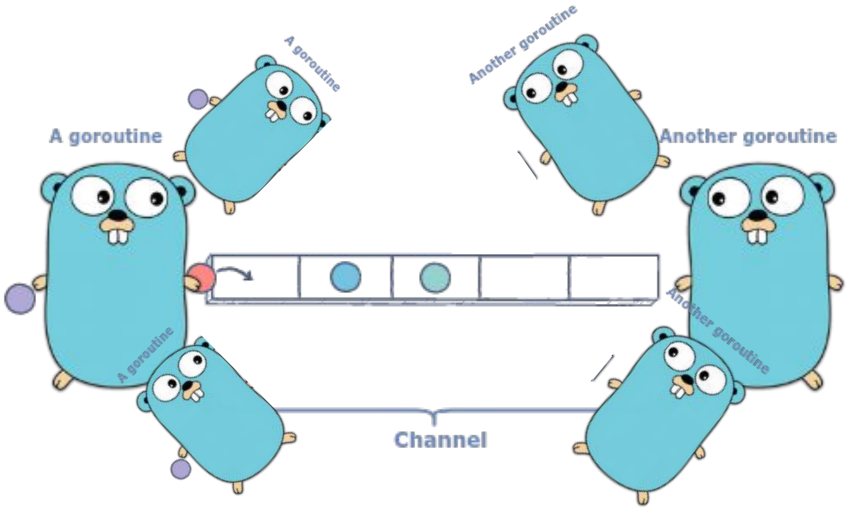
Why this tutorial?

Understanding how to approach problems from two perspectives

Shared Memory

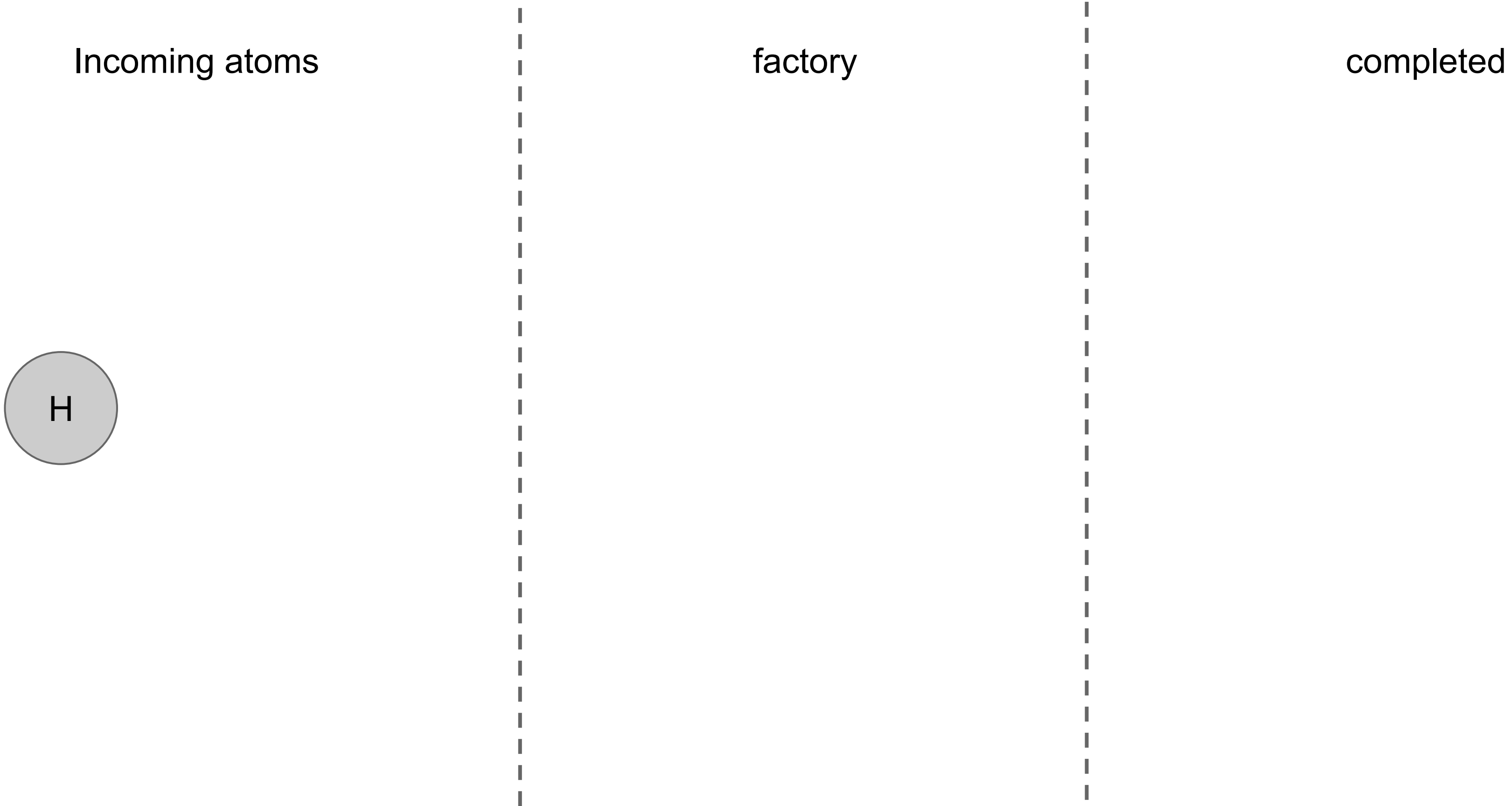


Channels / Distributed Memory

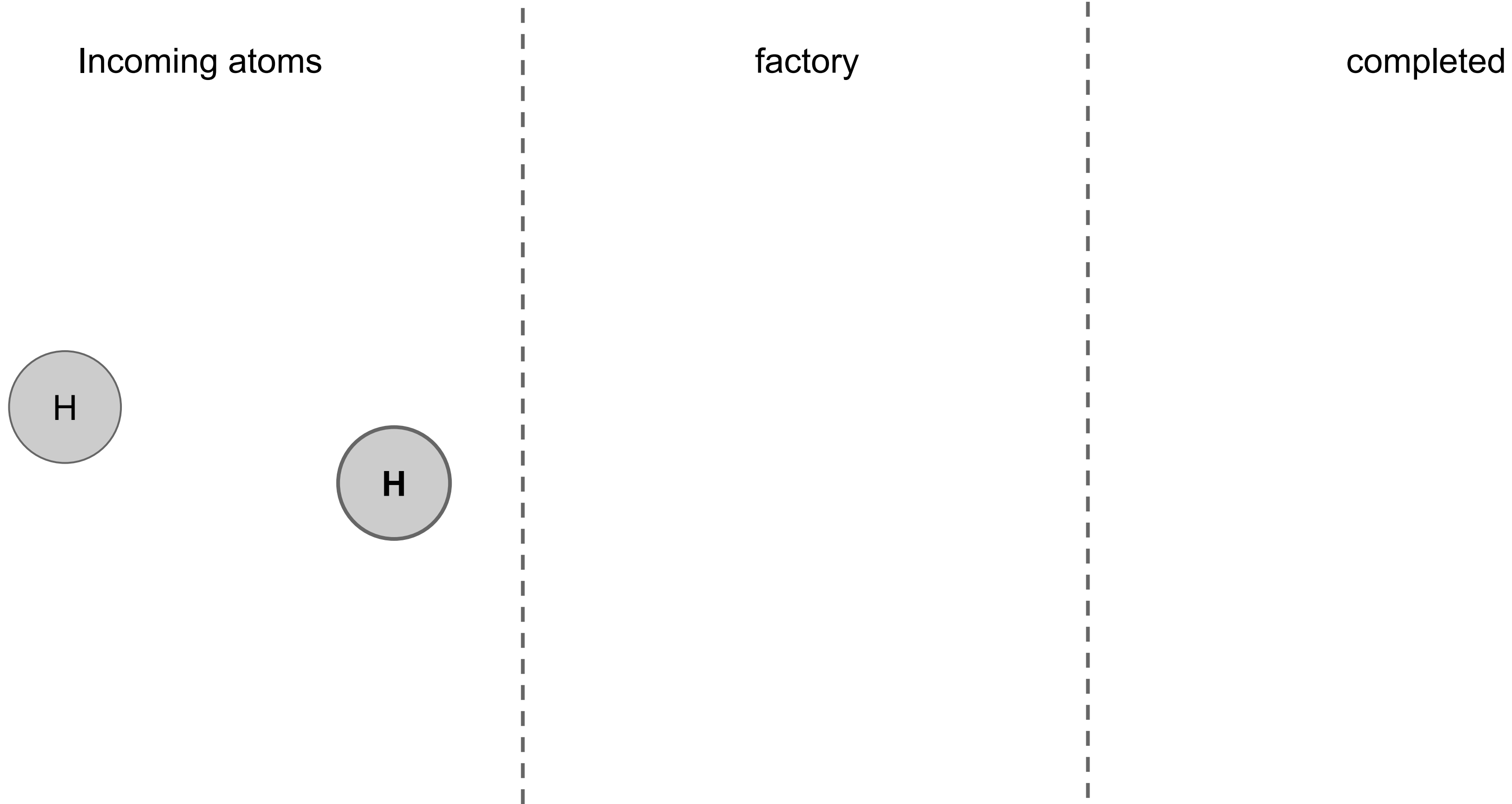


The H2O Problem: Shared Memory

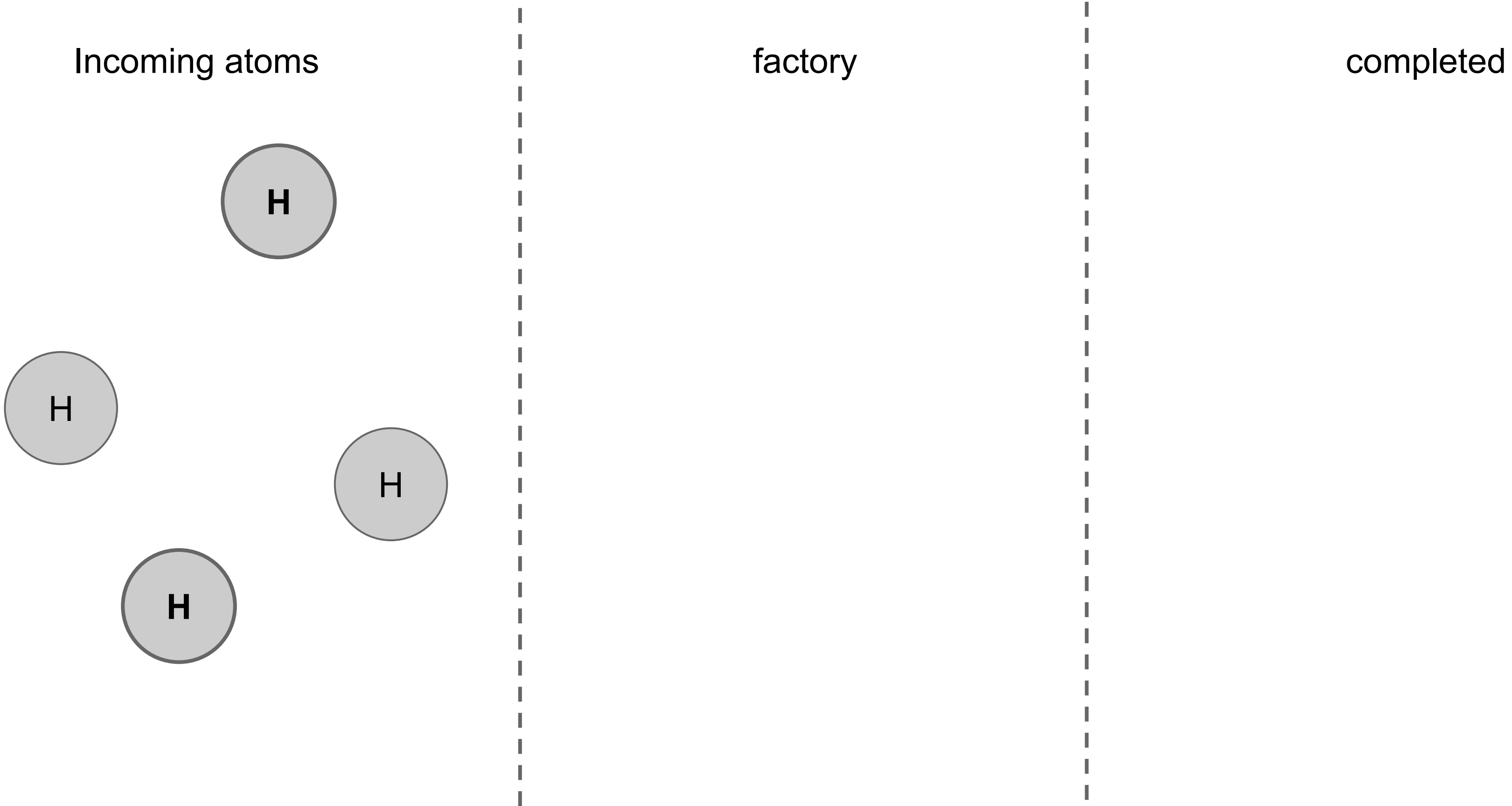
1. H2O Problem



1. H2O Problem



1. H2O Problem

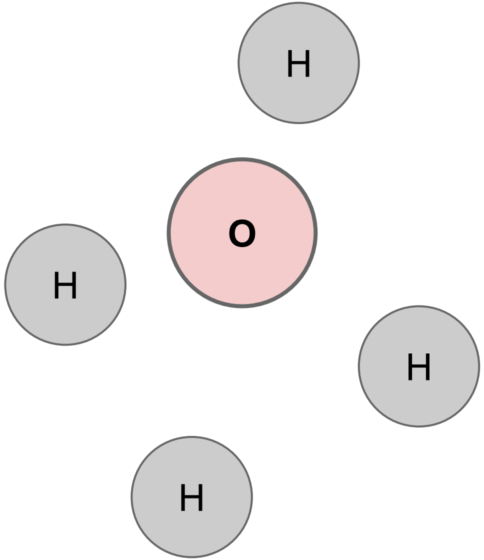


1. H2O Problem

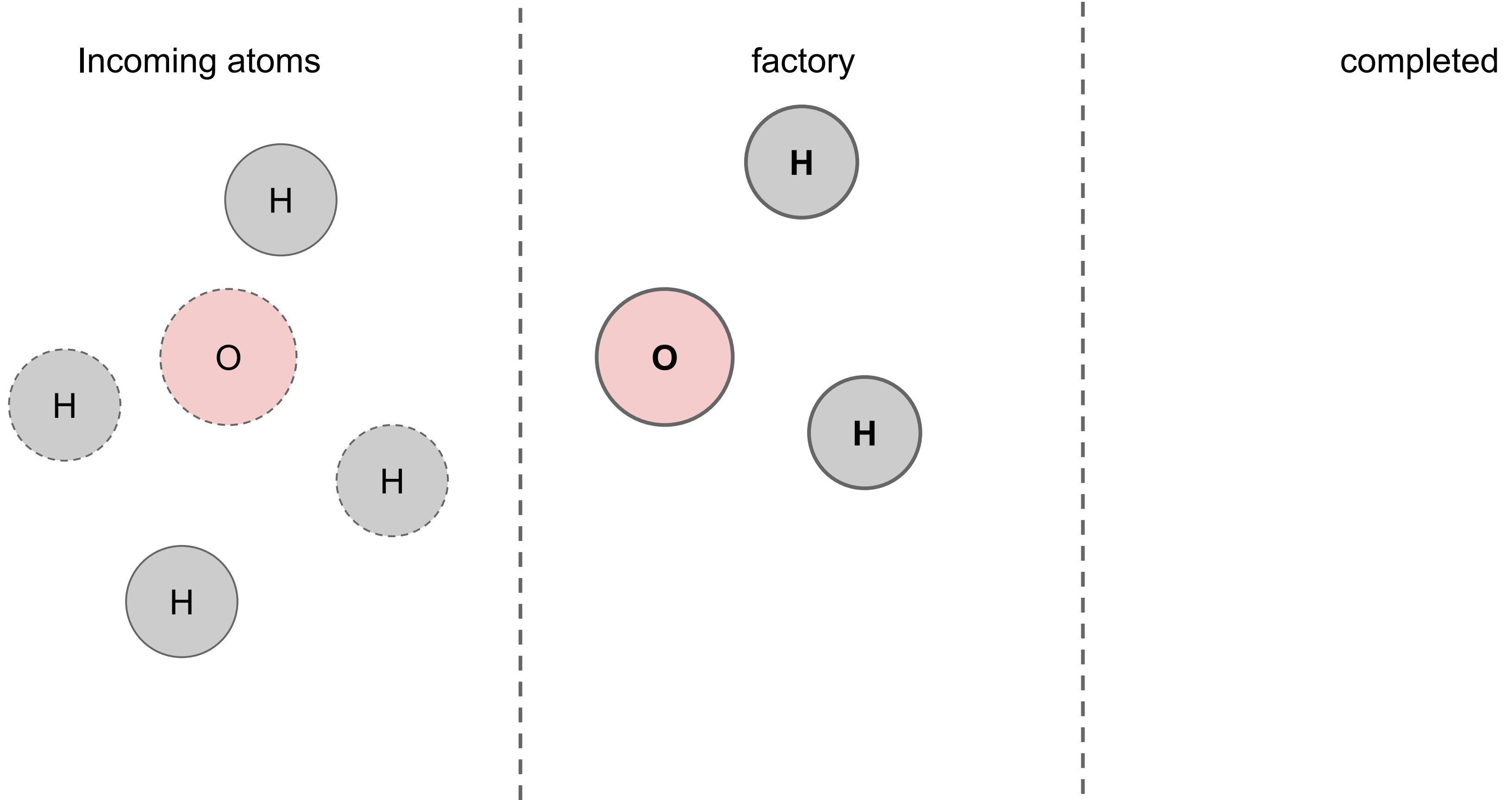
Incoming atoms

factory

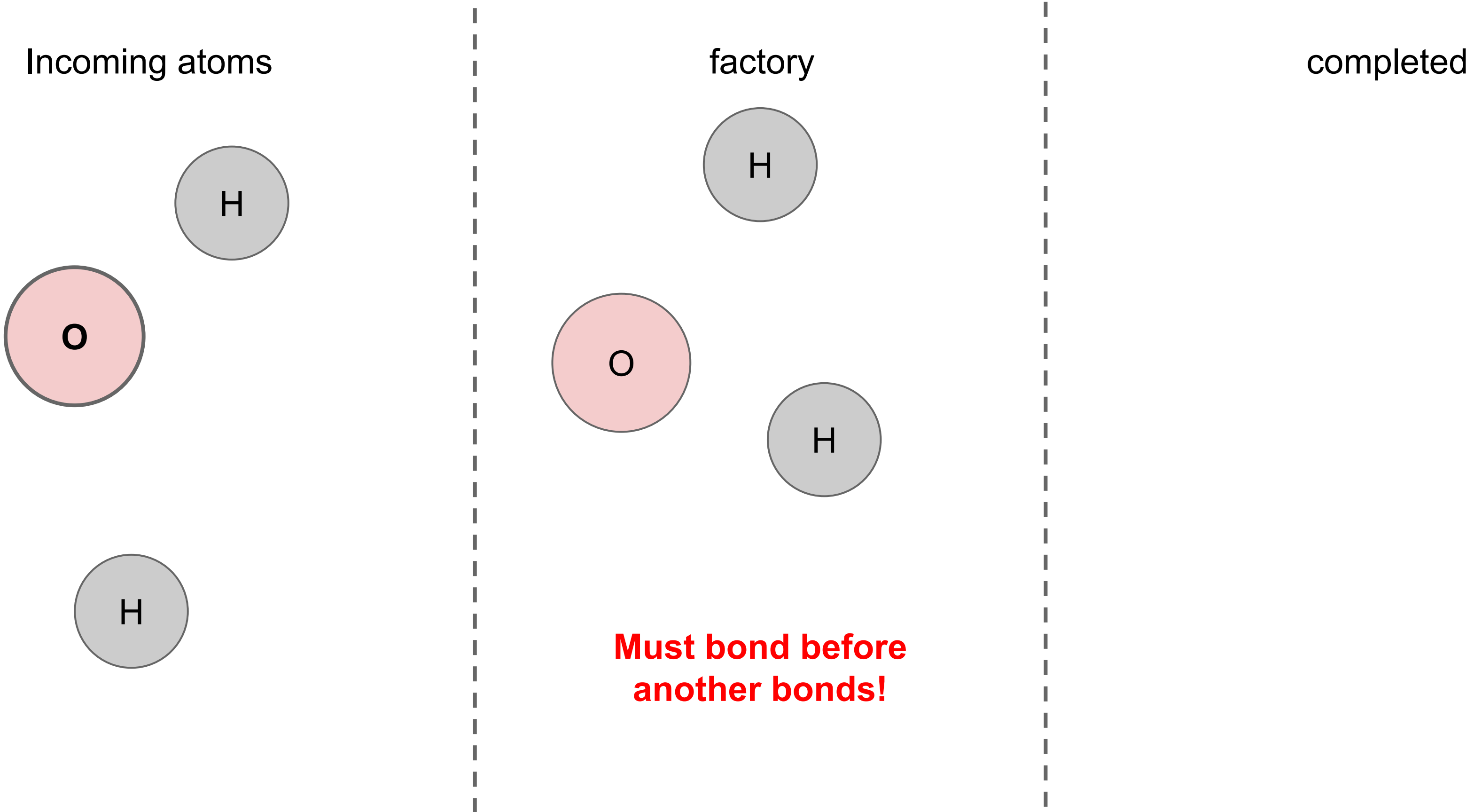
completed



1. H2O Problem



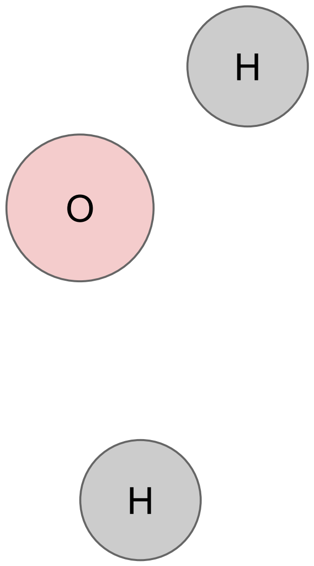
1. H2O Problem



Must bond before another bonds!

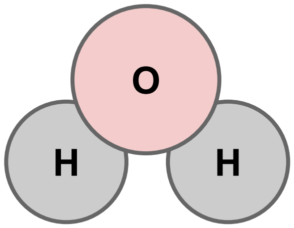
1. H2O Problem

Incoming atoms



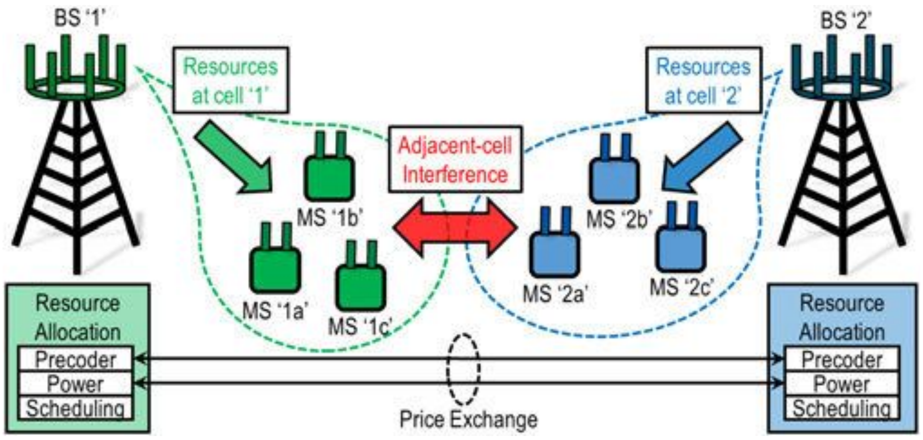
factory

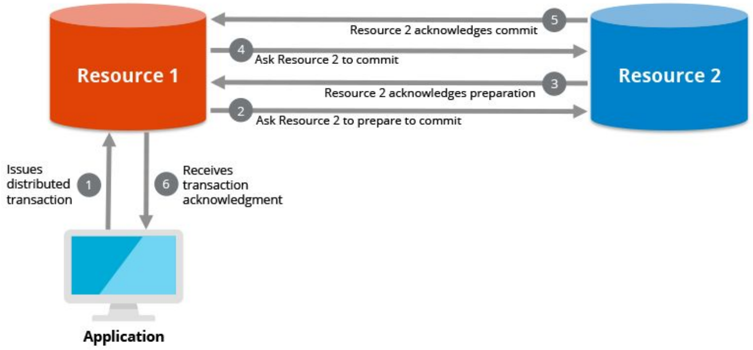
completed



Why this problem?

- Structurally similar to multiple real-world problems
- Distributed **resource allocation**
- Sister's Problems: Cigarette Smokers Problem, River Crossing Problem.





The H2O Problem: Shared Memory Barrier Solution

Solution: Barrier (Shared Memory)

```
struct WaterFactory {
  std::barrier<> barrier;
  WaterFactory() : barrier{3} {}

  void oxygen(void (*bond)()) {
    barrier.arrive_and_wait();
    bond();
  }

  void hydrogen(void (*bond)()) {
    barrier.arrive_and_wait();
    bond();
  }
};
```

What's the problem?

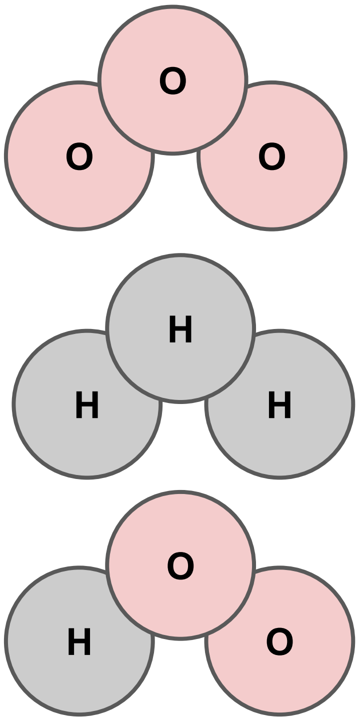
https://fsmbolt.comp.nus.edu.sg/z/KErbdj

Solution: Barrier (Shared Memory)

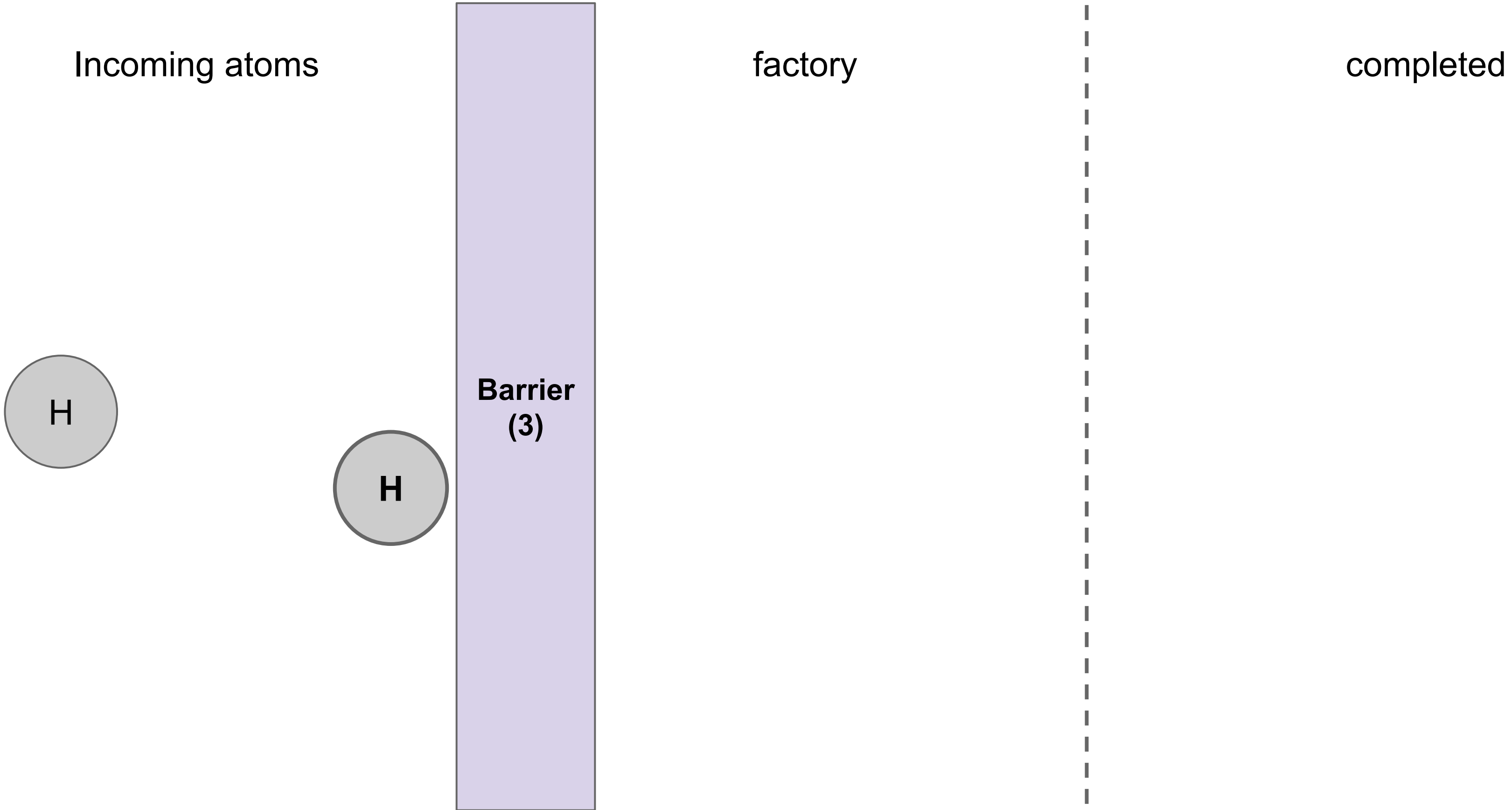
```
struct WaterFactory {
  std::barrier<> barrier;
  WaterFactory() : barrier{3} {}

  void oxygen(void (*bond)()) {
    barrier.arrive_and_wait();
    bond();
  }

  void hydrogen(void (*bond)()) {
    barrier.arrive_and_wait();
    bond();
  }
};
```

https://fsmbolt.comp.nus.edu.sg/z/KErbdj

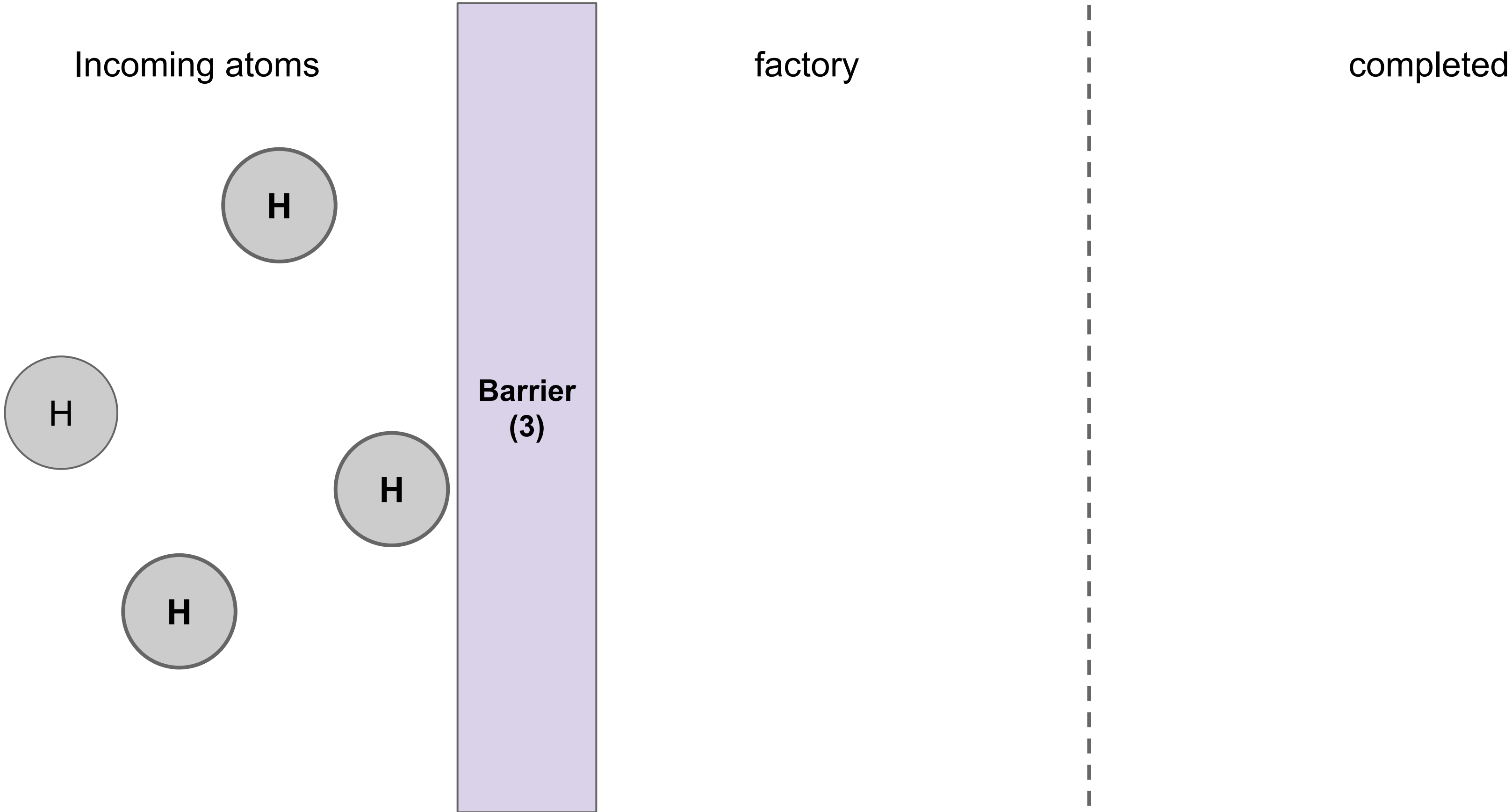
Any 3 atoms can bond!



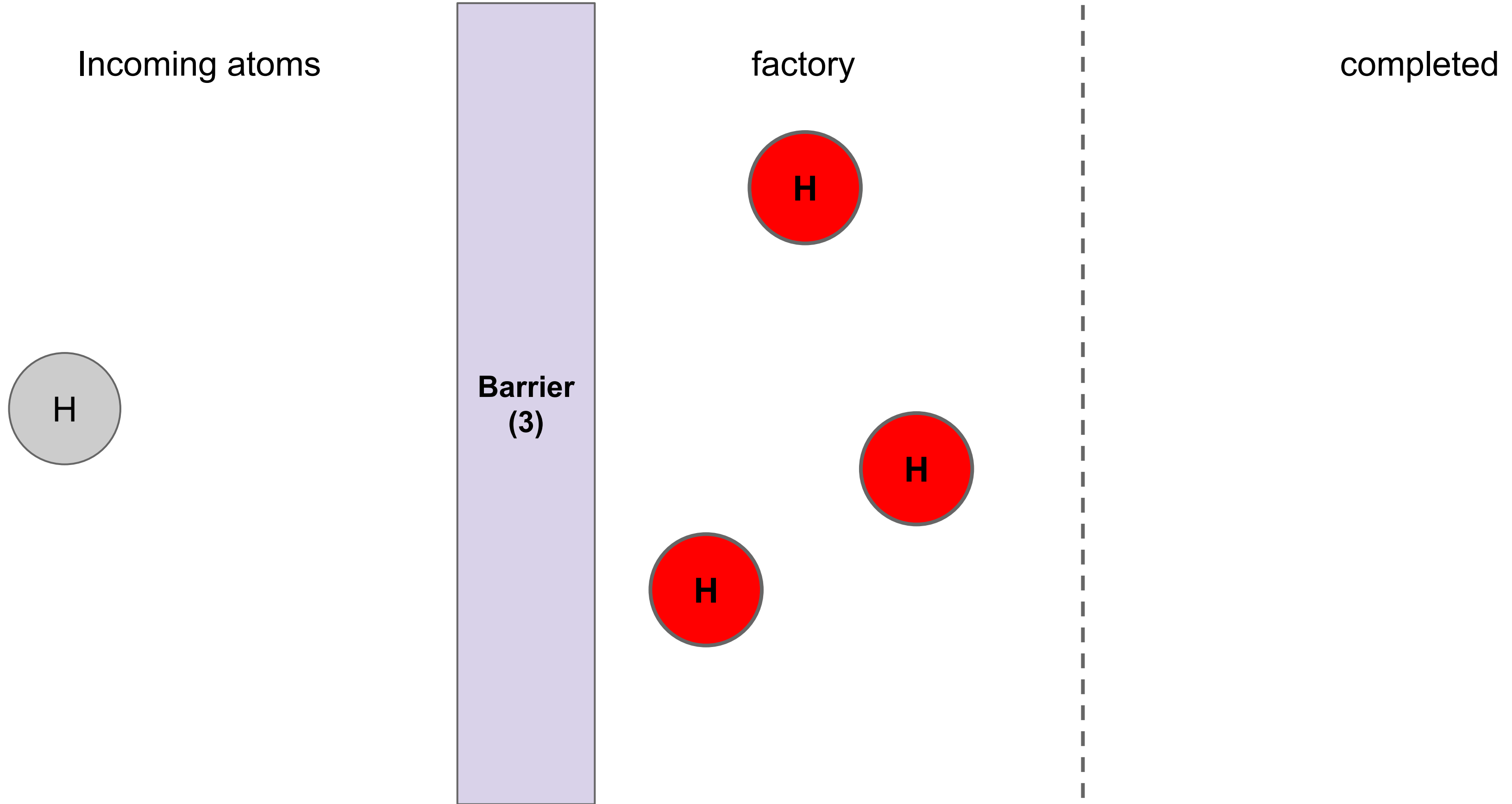
1. H2O Problem



1. H2O Problem



1. H2O Problem



The H2O Problem: Shared Memory Semaphore Solution

Solution: Semaphores (Extra)

```
struct WaterFactory {
  std::counting_semaphore oSem{1}, hSem{2};
  WaterFactory() {}

  void oxygen(void (*bond)()) {
    oSem.acquire();
    bond();
    oSem.release();
  }

  void hydrogen(void (*bond)()) {
    hSem.acquire();
    bond();
    hSem.release();
  }
};
```

https://fsmbolt.comp.nus.edu.sg/z/85Y7W1

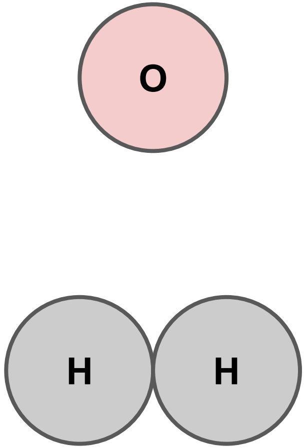
What non-H2O molecules are possible with this implementation (if any?) [p]

Solution: Semaphores (Extra)

"Free Radical" Molecules!

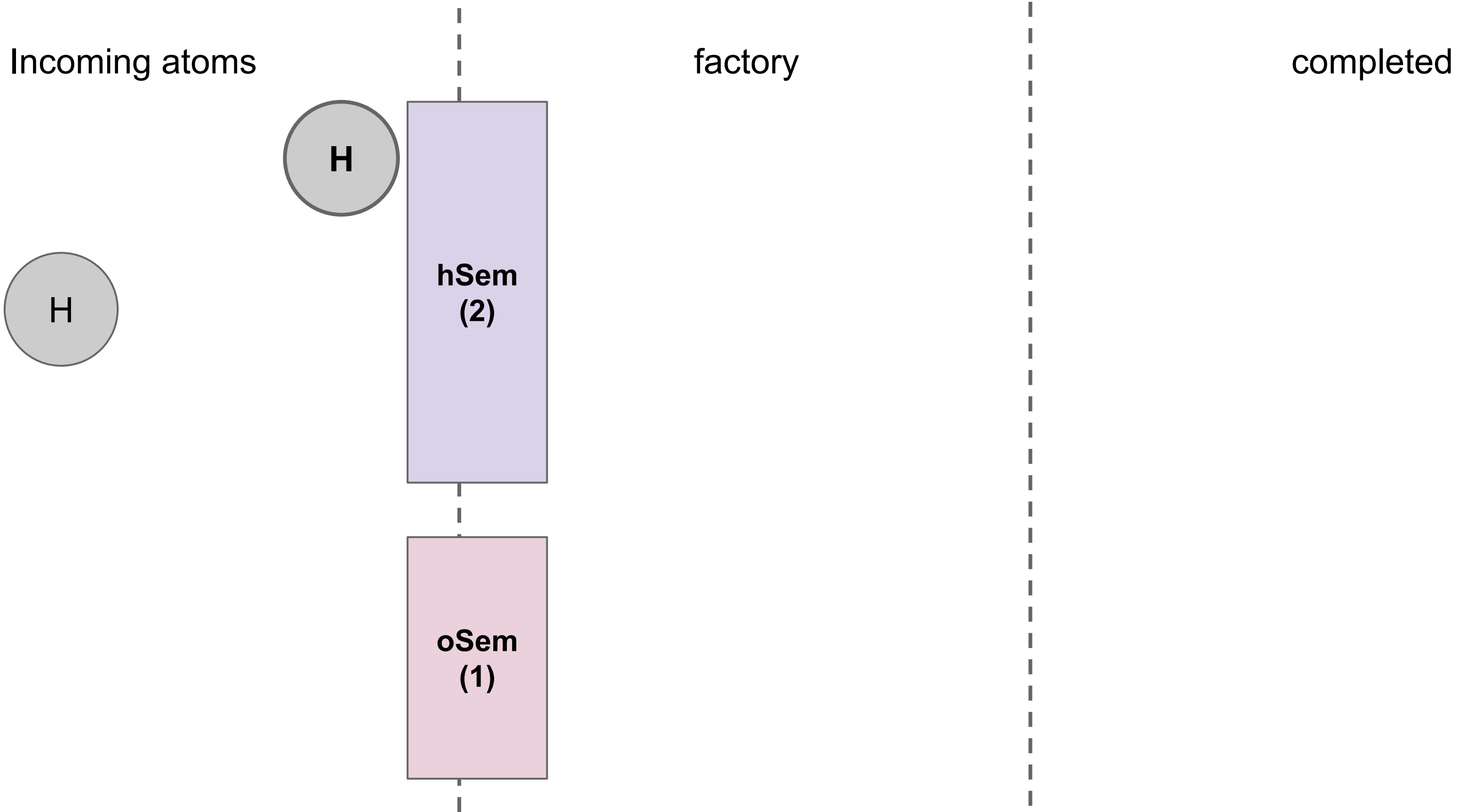
```
struct WaterFactory {
  std::counting_semaphore oSem{1}, hSem{2};
  WaterFactory() {}

  void oxygen(void (*bond)()) {
    oSem.acquire();
    bond();
    oSem.release();
  }

  void hydrogen(void (*bond)()) {
    hSem.acquire();
    bond();
    hSem.release();
  }
};
```

These functions are completely unsynchronized wrt each other

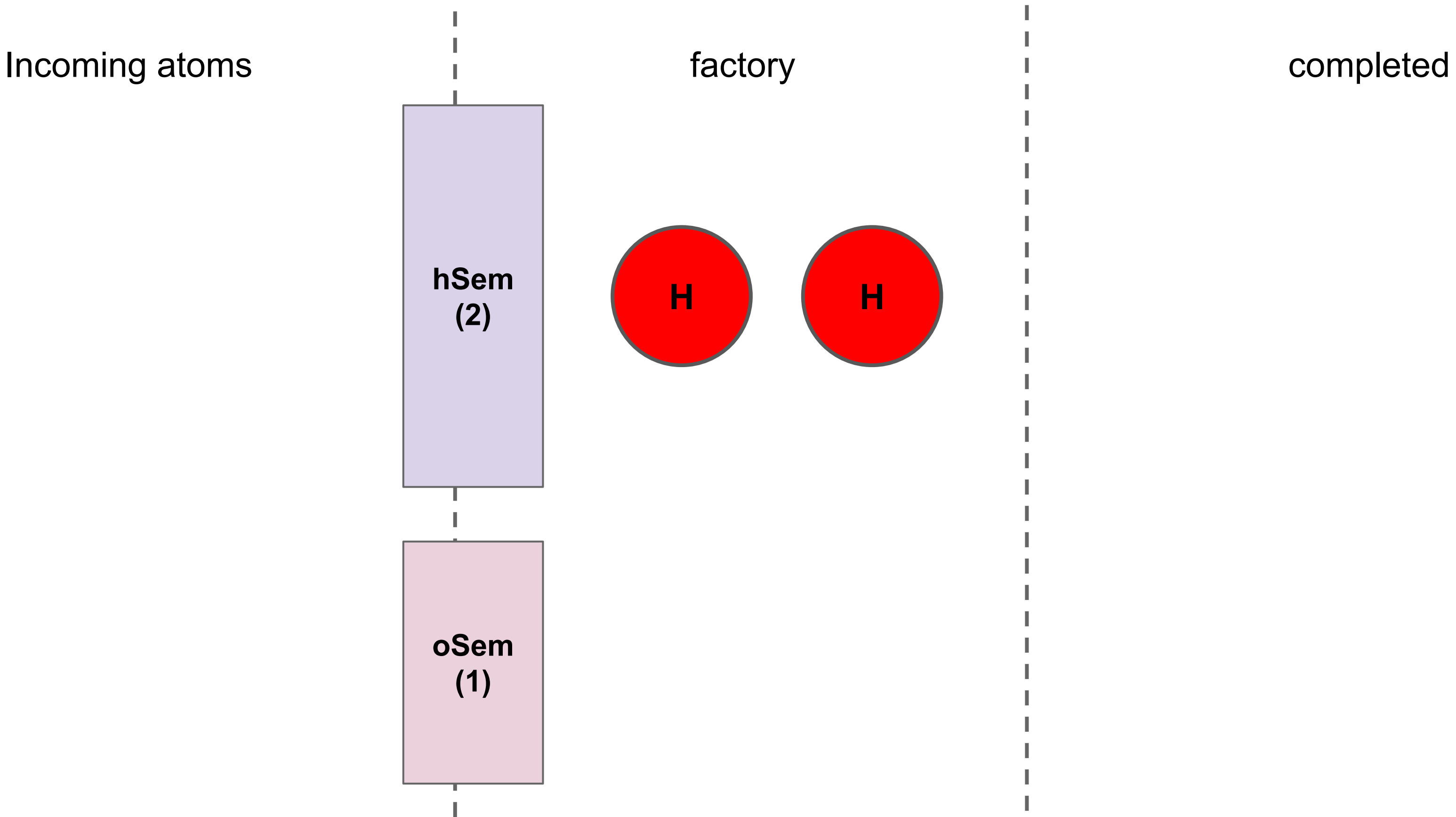


https://fsmbolt.comp.nus.edu.sg/z/85Y7W1

1. H2O Problem



1. H2O Problem



The H2O Problem: Shared Memory Semaphores + Barrier Solution

Solution: Semaphores + Barrier

```
struct WaterFactory {
  std::counting_semaphore oSem{1}, hSem{2};
  std::barrier<> barrier;
  WaterFactory() : barrier{3} {}

  void oxygen(void (*bond)()) {
    oSem.acquire();
    barrier.arrive_and_wait();
    bond();
    oSem.release();
  }

  void hydrogen(void (*bond)()) {
    hSem.acquire();
    barrier.arrive_and_wait();
    bond();
    hSem.release();
  }
};
```

https://fsmbolt.comp.nus.edu.sg/z/hh1sbP

Is there a problem?

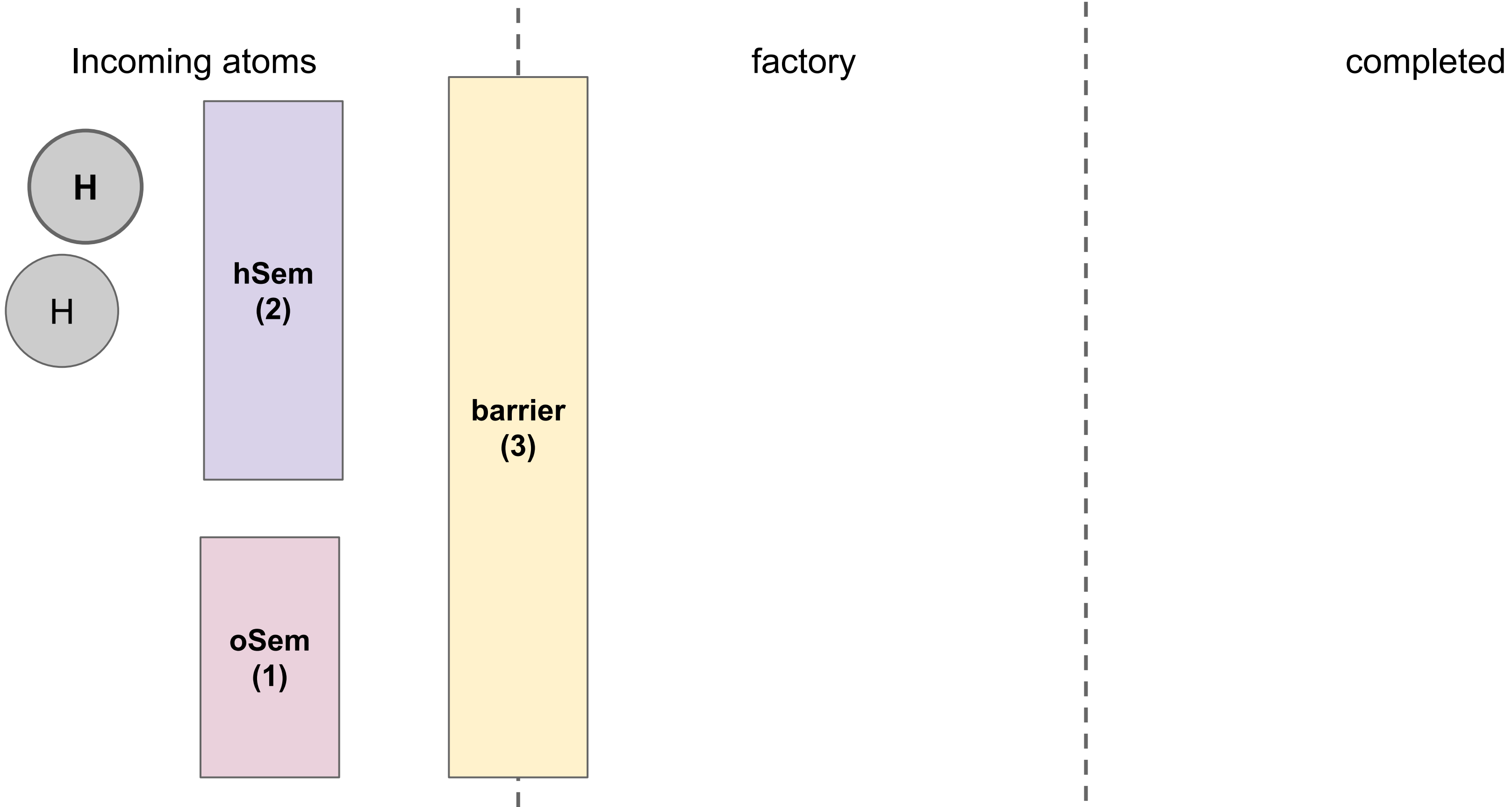
Solution: Semaphores + Barrier

```
struct WaterFactory {
  std::counting_semaphore oSem{1}, hSem{2};
  std::barrier<> barrier;
  WaterFactory() : barrier{3} {}

  void oxygen(void (*bond)()) {
    oSem.acquire();
    barrier.arrive_and_wait();
    bond();
    oSem.release();
  }

  void hydrogen(void (*bond)()) {
    hSem.acquire();
    barrier.arrive_and_wait();
    bond();
    hSem.release();
  }
};
```

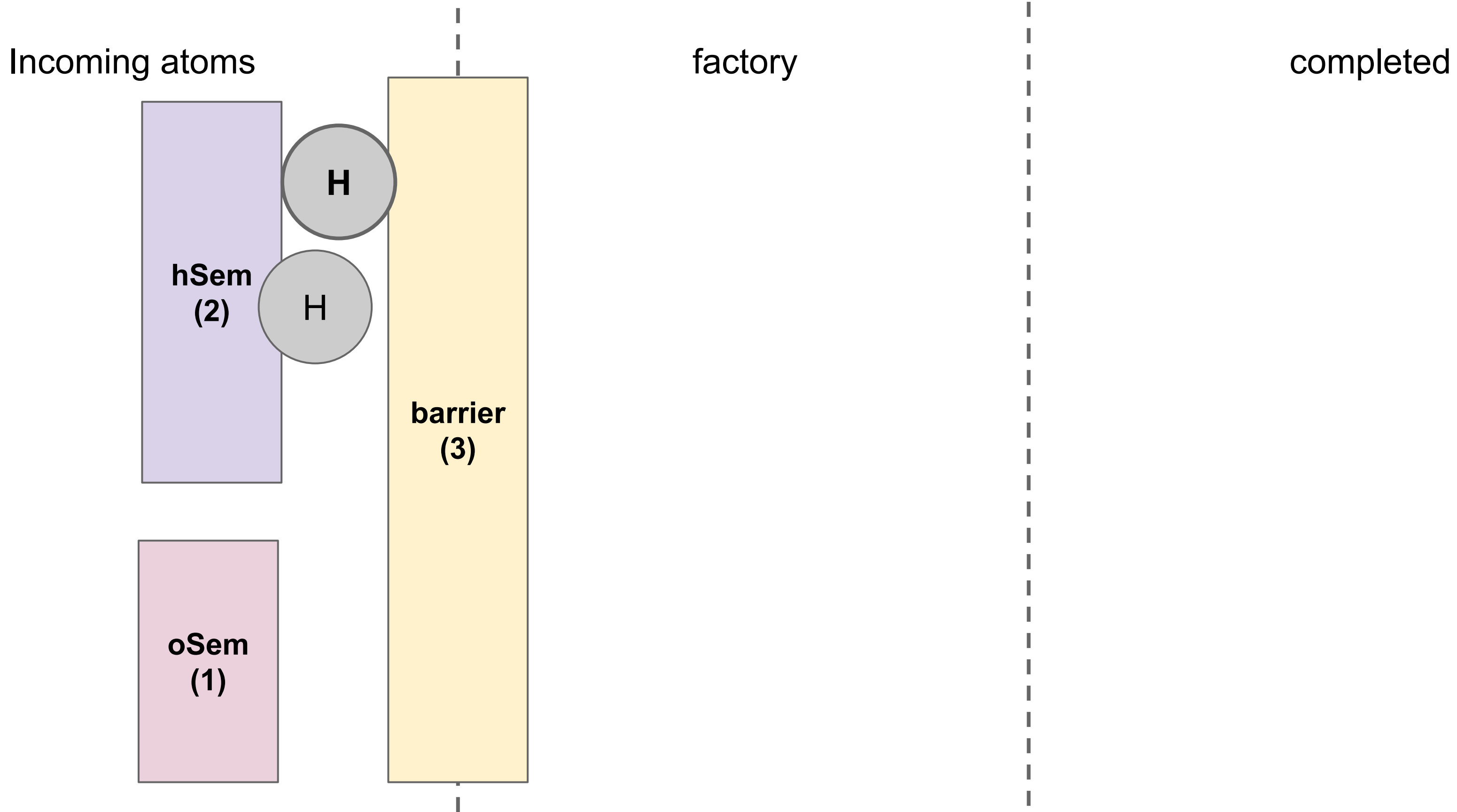
https://fsmbolt.comp.nus.edu.sg/z/hh1sbP

No more!

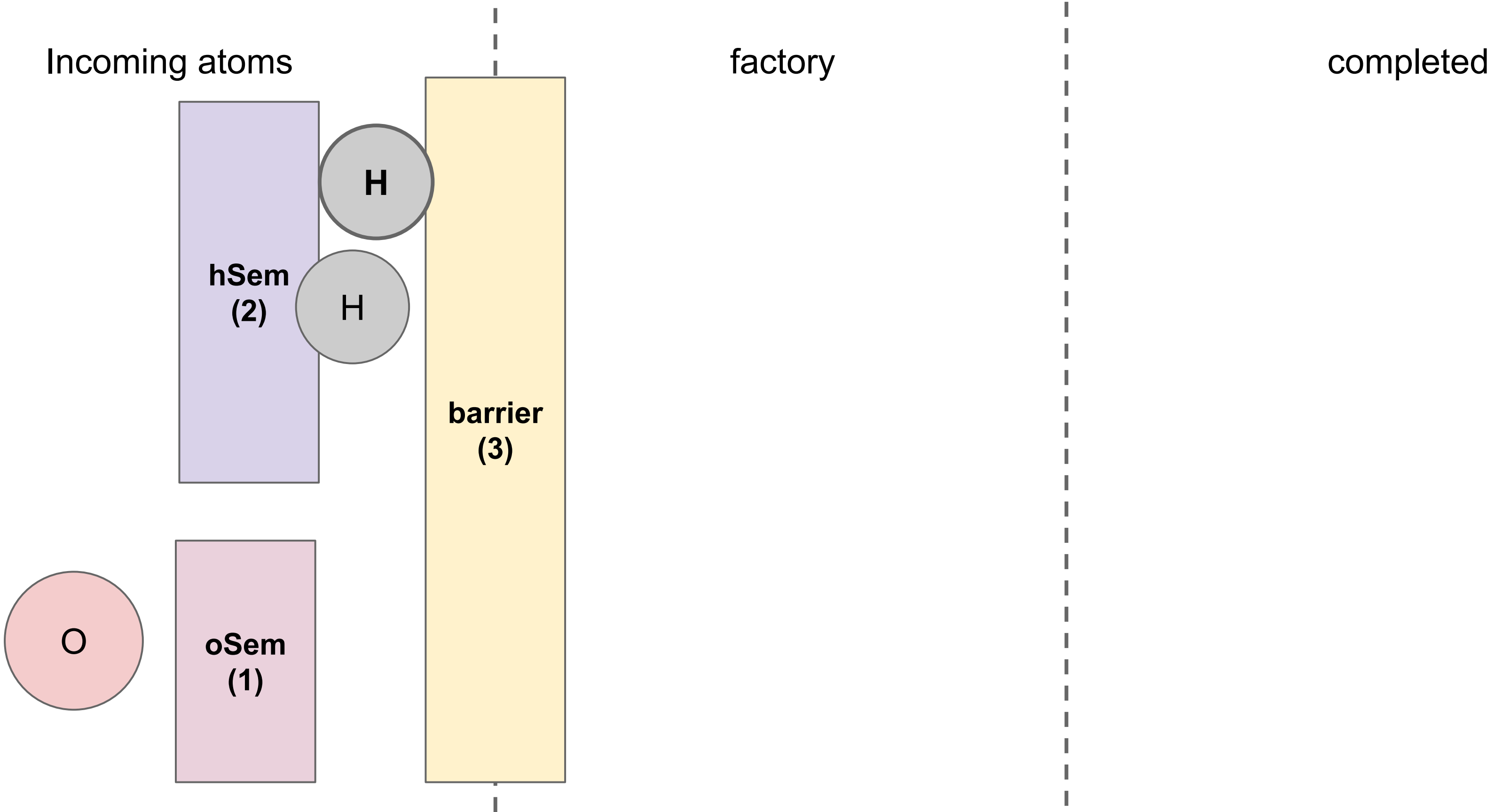
1. H2O Problem



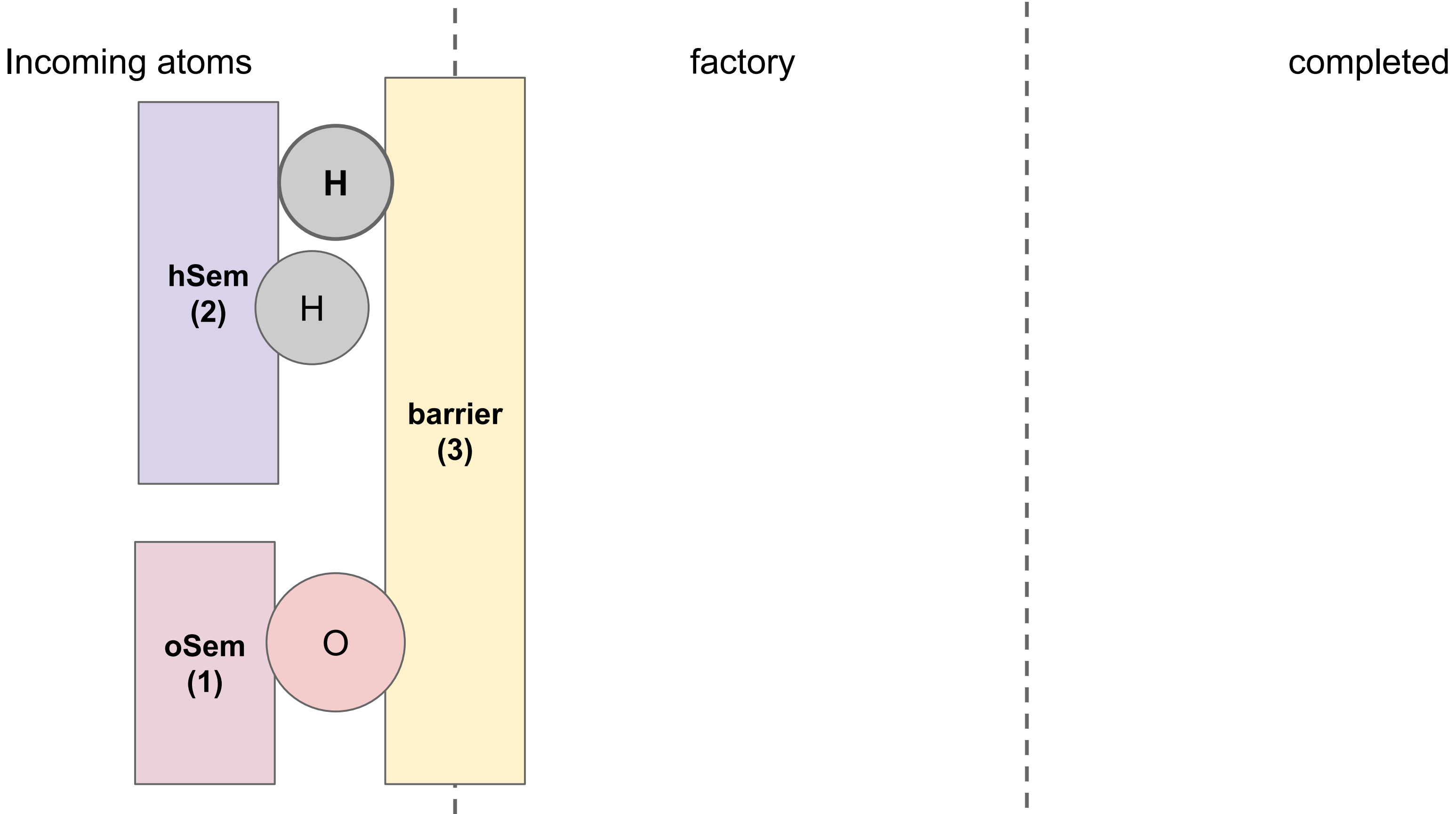
1. H2O Problem



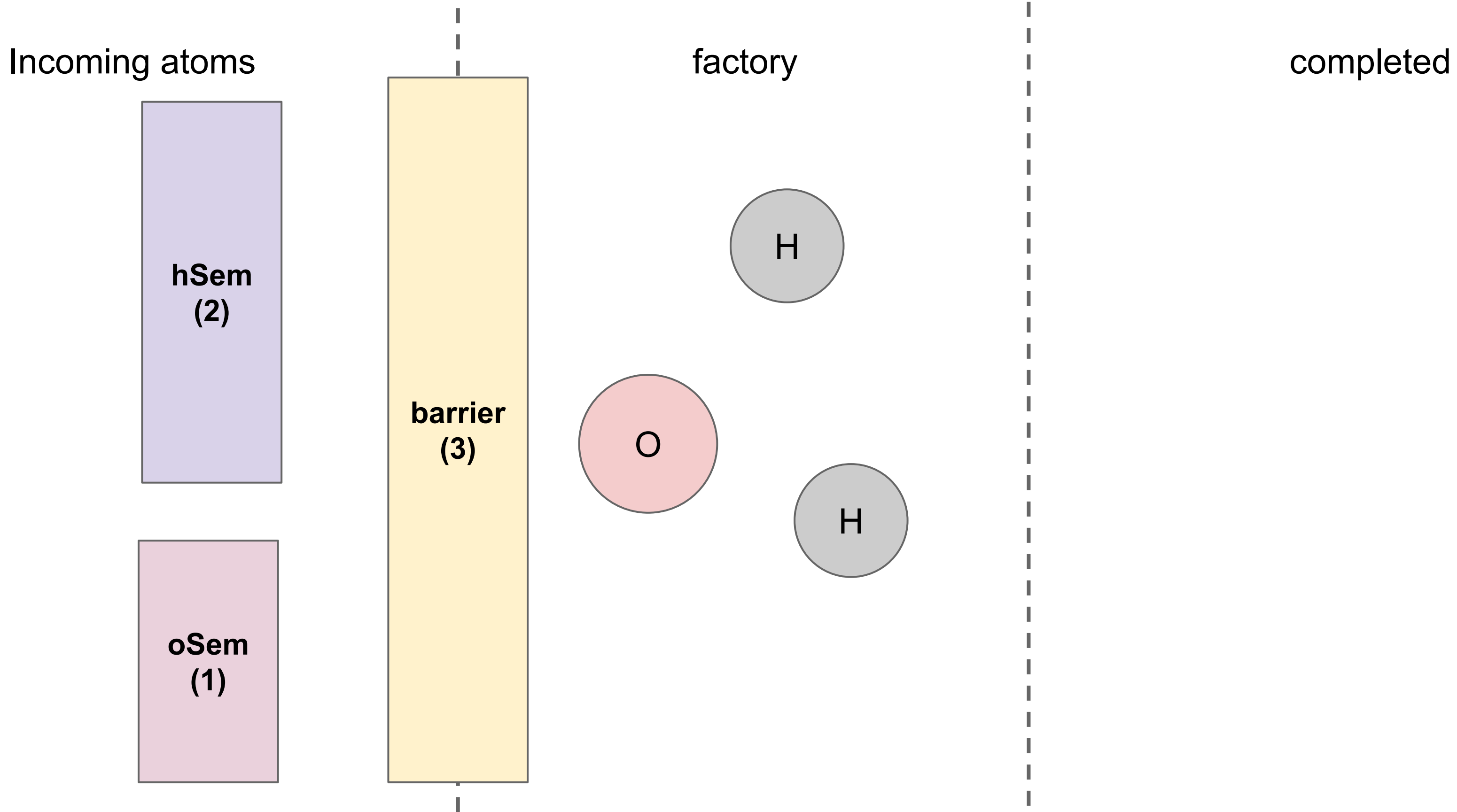
1. H2O Problem



1. H2O Problem



1. H2O Problem



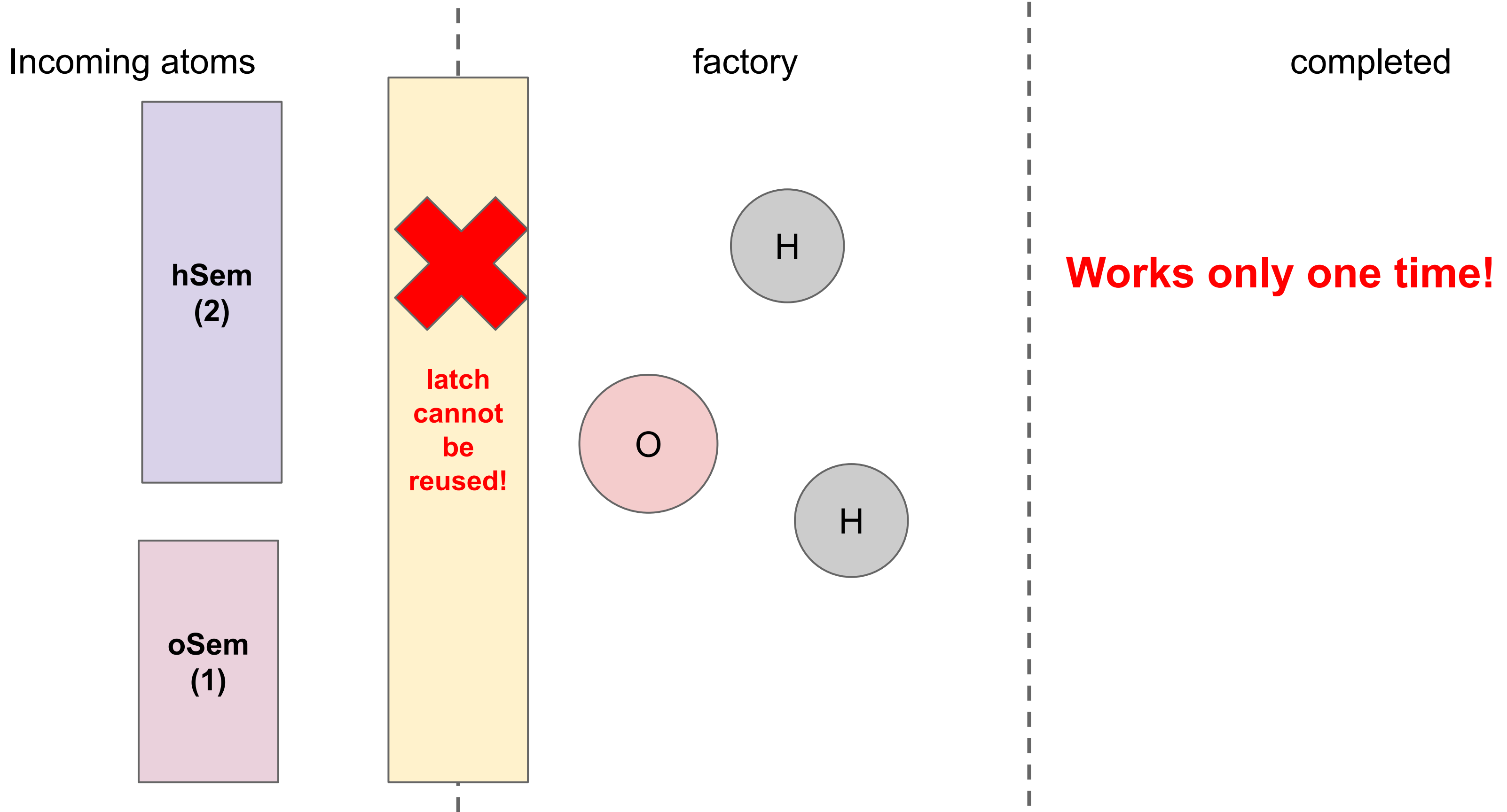
The H2O Problem: Shared Memory Semaphores + Latch Solution

Solution: Semaphores + Latch

```
struct WaterFactory {
  std::counting_semaphore oSem{1}, hSem{2};
  std::latch<> latch;
  WaterFactory() : latch{3} {}

  void oxygen(void (*bond)()) {
    oSem.acquire();
    latch.arrive_and_wait();
    bond();
    oSem.release();
  }

  void hydrogen(void (*bond)()) {
    hSem.acquire();
    latch.arrive_and_wait();
    bond();
    hSem.release();
  }
};
```

What if we use a latch? [p]

The `latch` class is a downward counter of type `std::ptrdiff_t` which can be used to synchronize threads. The value of the counter is initialized on creation. Threads may block on the latch until the counter is decremented to zero. There is no possibility to increase or reset the counter, which makes the latch a single-use barrier.



The H2O Problem: Message Passing

Go: Daemon Solution

```go
// Step 1: (Precommit)
//          Receive arrival requets from 2 hydrogen and 1 oxygen atoms
h1 := <-wfd.precomH
h2 := <-wfd.precomH
o := <-wfd.precomO

// Step 2: (Commit)
//          Tell the 3 atoms to start bonding
h1 <- struct{}{}
h2 <- struct{}{}
o <- struct{}{}

// Step 3: (Postcommit)
//          Wait until the 3 atoms have finished before looping
// We re-use the same communication channel as (Commit)
<-h1
<-h2
<-o
```

```go
type WaterFactoryWithDaemon struct {
    // Channels for atoms to send their arrival requests
    precomH chan chan struct{}
    precomO chan chan struct{}
}
```

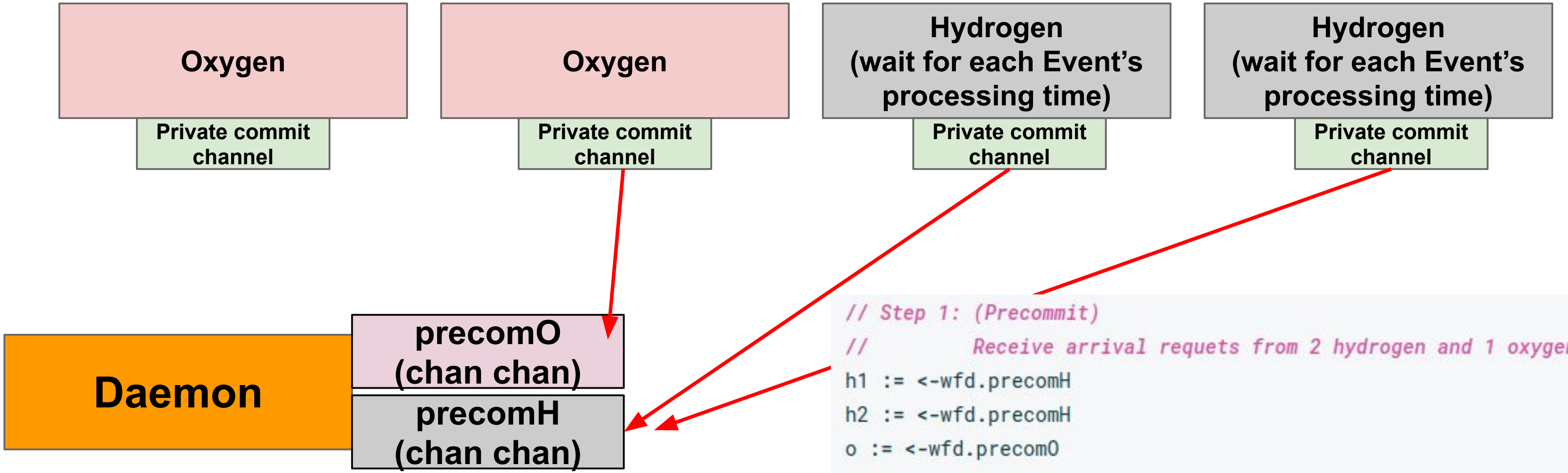
```go
func (wfd *WaterFactoryWithDaemon) hydrogen(bond func()) {
    commit := make(chan struct{}) // Step 1: Create private communication channel
    wfd.precomH <- commit         // Step 2: (Precommit)
    <-commit                      // Step 3: (Commit)
    bond()                        // Step 4: Bond
    commit <- struct{}{}          // Step 5: (Postcommit)
}
```

The H2O Problem: Message Passing Daemon Solution

Go: Daemon Solution

```go
func (wfd *WaterFactoryWithDaemon) hydrogen(bond func()) {
    commit := make(chan struct{}) // Step 1: Create private communication channel
    wfd.precomH <- commit          // Step 2: (Precommit)
    <-commit                       // Step 3: (Commit)
    bond()                         // Step 4: Bond
    commit <- struct{}{}           // Step 5: (Postcommit)
}
```

- Daemon waits for 2H and 1O channel in the respective chanchans

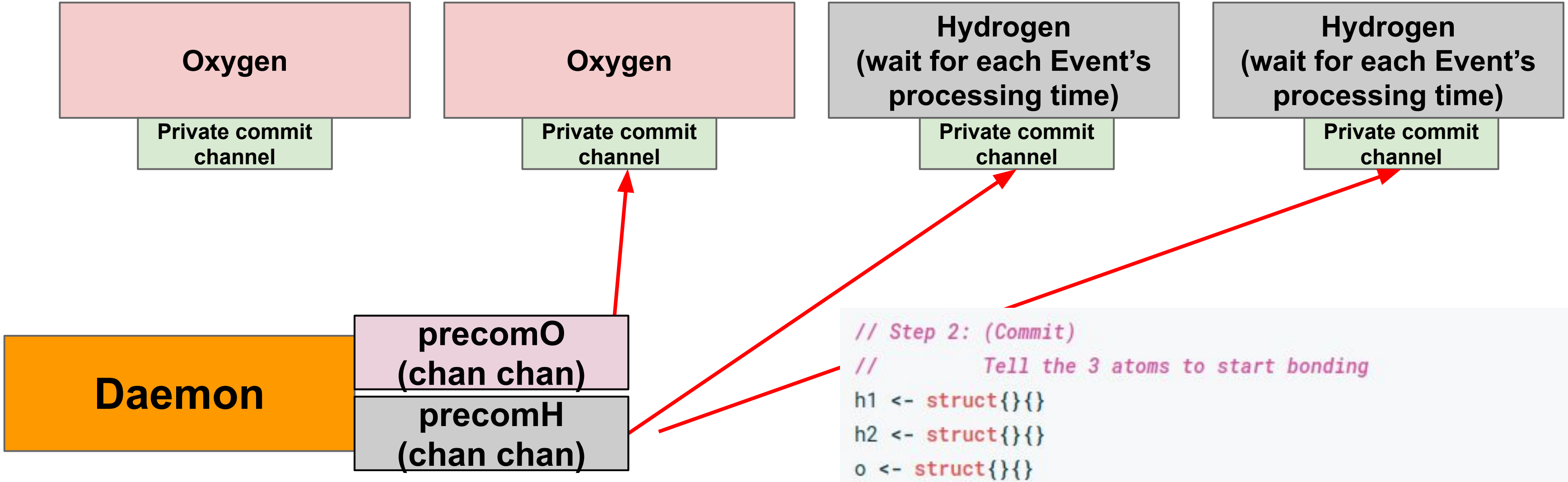



```go
// Step 1: (Precommit)
//         Receive arrival requets from 2 hydrogen and 1 oxygen atoms
h1 := <-wfd.precomH
h2 := <-wfd.precomH
o := <-wfd.precomO
```

Go: Daemon Solution

```go
func (wfd *WaterFactoryWithDaemon) hydrogen(bond func()) {
    commit := make(chan struct{}) // Step 1: Create private communication channel
    wfd.precomH <- commit         // Step 2: (Precommit)
    <-commit                      // Step 3: (Commit)
    bond()                        // Step 4: Bond
    commit <- struct{}{}          // Step 5: (Postcommit)
}
```

- Pushes a value to the 2H and 1O channel that it saw (unblocking them)

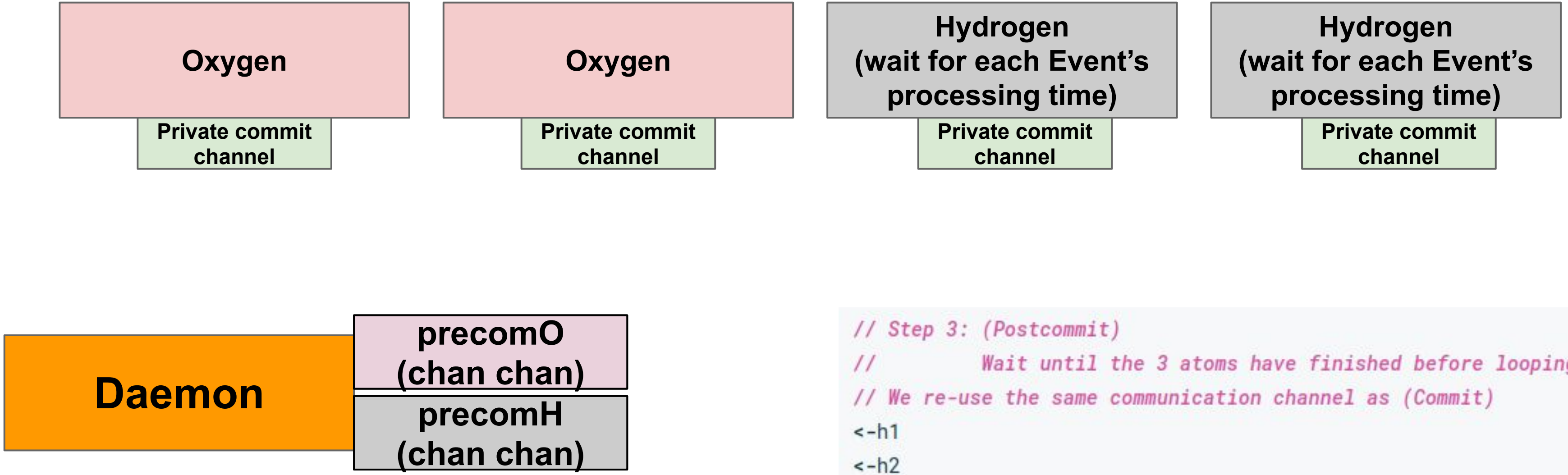



```go
// Step 2: (Commit)
//          Tell the 3 atoms to start bonding
h1 <- struct{}{}
h2 <- struct{}{}
o <- struct{}{}
```

Go: Daemon Solution

```
func (wfd *WaterFactoryWithDaemon) hydrogen(bond func()) {
    commit := make(chan struct{}) // Step 1: Create private communication channel
    wfd.precomH <- commit         // Step 2: (Precommit)
    <-commit                      // Step 3: (Commit)
    bond()                        // Step 4: Bond
    commit <- struct{}{}          // Step 5: (Postcommit)
}
```

- The 2H, 1O start bond(), Daemon is waiting for a message on the channels…

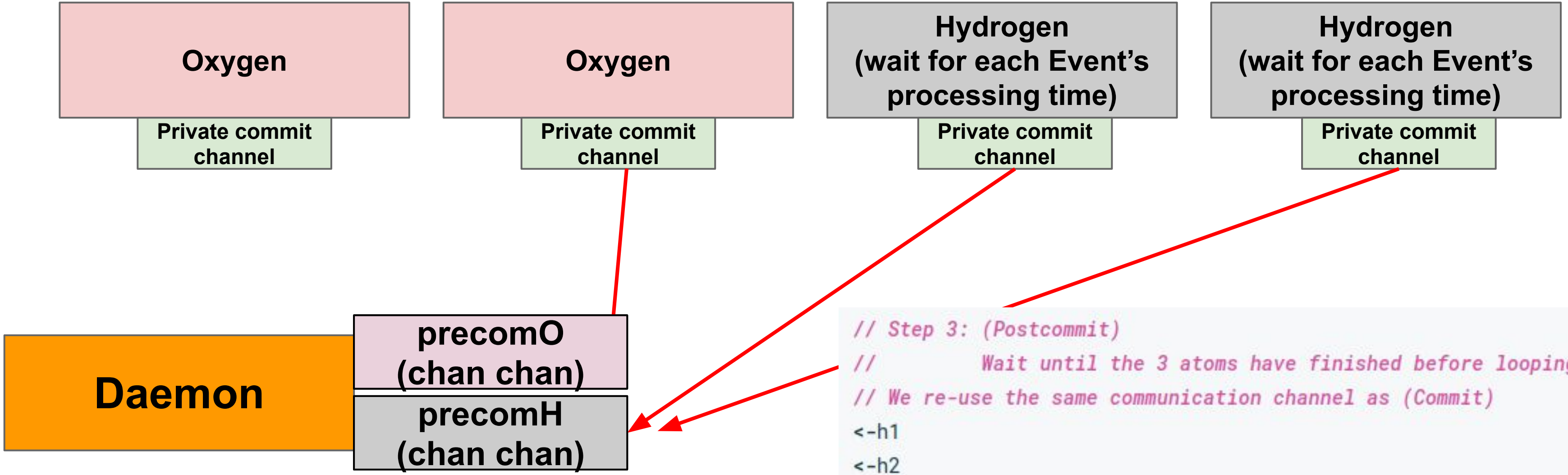



```
// Step 3: (Postcommit)
//         Wait until the 3 atoms have finished before looping
// We re-use the same communication channel as (Commit)
<-h1
<-h2
<-o
```

Go: Daemon Solution

```go
func (wfd *WaterFactoryWithDaemon) hydrogen(bond func()) {
    commit := make(chan struct{}) // Step 1: Create private communication channel
    wfd.precomH <- commit         // Step 2: (Precommit)
    <-commit                      // Step 3: (Commit)
    bond()                        // Step 4: Bond
    commit <- struct{}{}          // Step 5: (Postcommit)
}
```

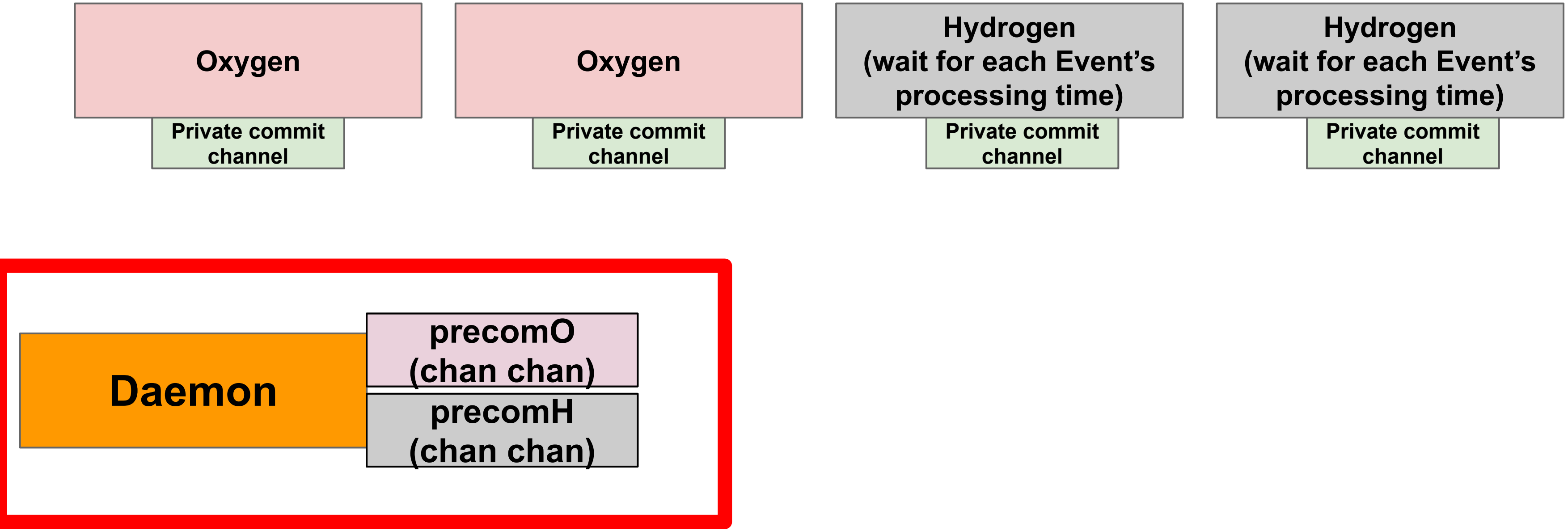
- Daemon receives messages that the 2H 1O are done, **go back to start and do it all over again!**




```go
// Step 3: (Postcommit)
//         Wait until the 3 atoms have finished before looping
// We re-use the same communication channel as (Commit)
<-h1
<-h2
<-o
```

Go: Daemon Solution

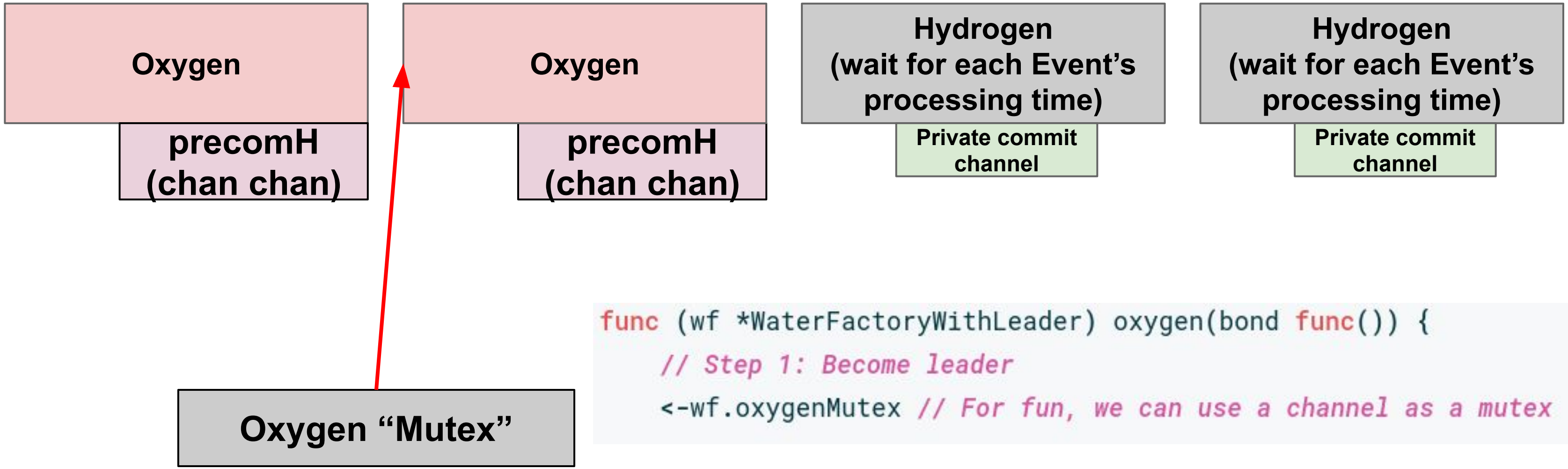
- Problems: **single point of failure** + **bottleneck** + **memory management**



The H2O Problem: Message Passing Oxygen Leader Solution

Go: Oxygen Leader Soln

- Oxygen atom tries to become “leader” through a mutex-like idea (either a channel with 1 capacity, or actual mutex

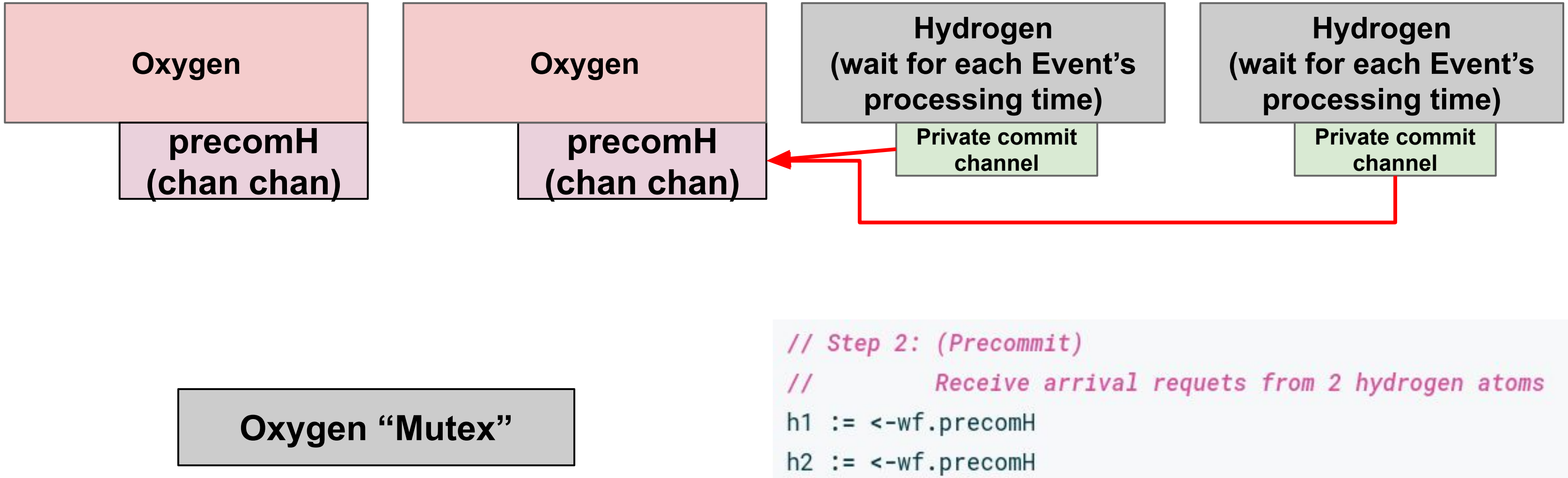



```go
func (wf *WaterFactoryWithLeader) oxygen(bond func()) {
    // Step 1: Become leader
    <-wf.oxygenMutex // For fun, we can use a channel as a mutex
```

Go: Oxygen Leader Soln

```go
func (wf *WaterFactoryWithLeader) hydrogen(bond func()) {
    commit := make(chan struct{}) // Step 1: Create private communication channel
    wf.precomH <- commit          // Step 2: (Precommit)
    <-commit                      // Step 3: (Commit)
    bond()                        // Step 4: Bond
    commit <- struct{}{}          // Step 5: (Postcommit)
}
```

- Oxygen waits for 2 hydrogens to show up

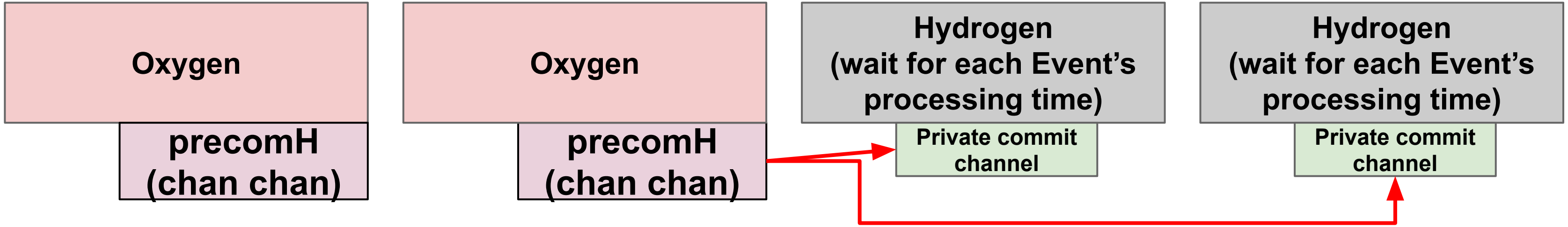



```go
// Step 2: (Precommit)
//         Receive arrival requets from 2 hydrogen atoms
h1 := <-wf.precomH
h2 := <-wf.precomH
```

Go: Oxygen Leader Soln

```go
func (wf *WaterFactoryWithLeader) hydrogen(bond func()) {
    commit := make(chan struct{}) // Step 1: Create private communication channel
    wf.precomH <- commit          // Step 2: (Precommit)
    <-commit                      // Step 3: (Commit)
    bond()                        // Step 4: Bond
    commit <- struct{}{}          // Step 5: (Postcommit)
}
```

- Oxygen unblocks both hydrogens

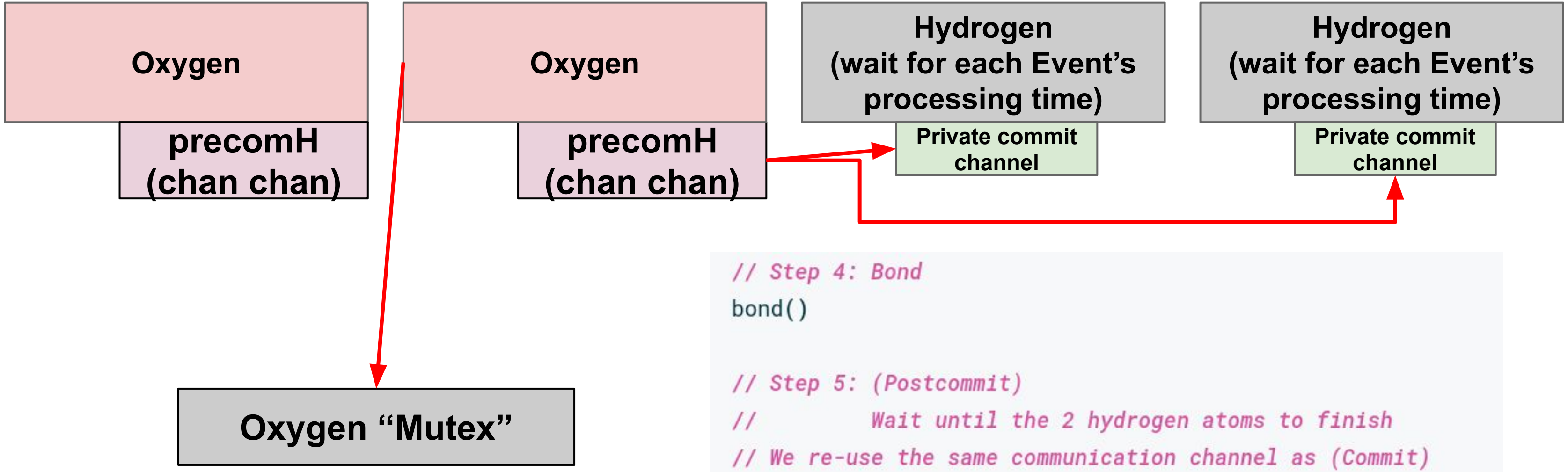



```go
// Step 3: (Commit)
//         Tell the 2 hydrogen atoms to start bonding
h1 <- struct{}{}
h2 <- struct{}{}
```

Go: Oxygen Leader Soln

```go
func (wf *WaterFactoryWithLeader) hydrogen(bond func()) {
    commit := make(chan struct{}) // Step 1: Create private communication channel
    wf.precomH <- commit          // Step 2: (Precommit)
    <-commit                      // Step 3: (Commit)
    bond()                        // Step 4: Bond
    commit <- struct{}{}          // Step 5: (Postcommit)
}
```

- They all bond, then hydrogens write back to channel
- Oxygen steps down as leader by “releasing mutex”

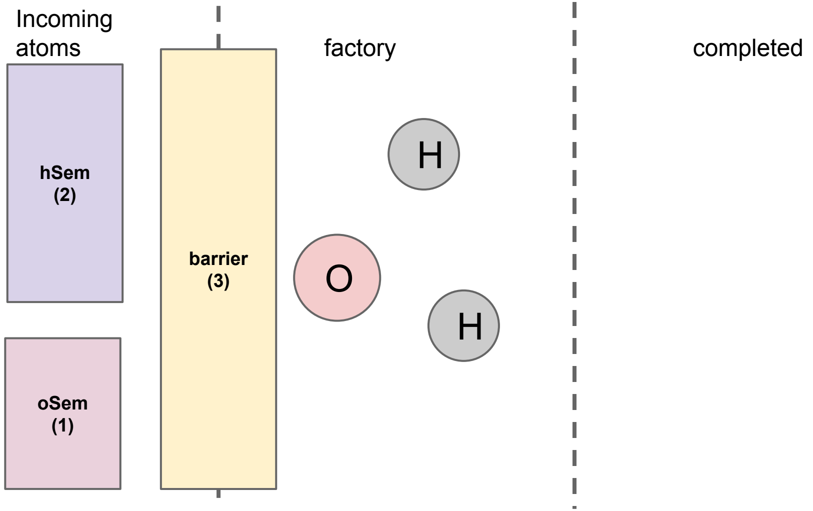


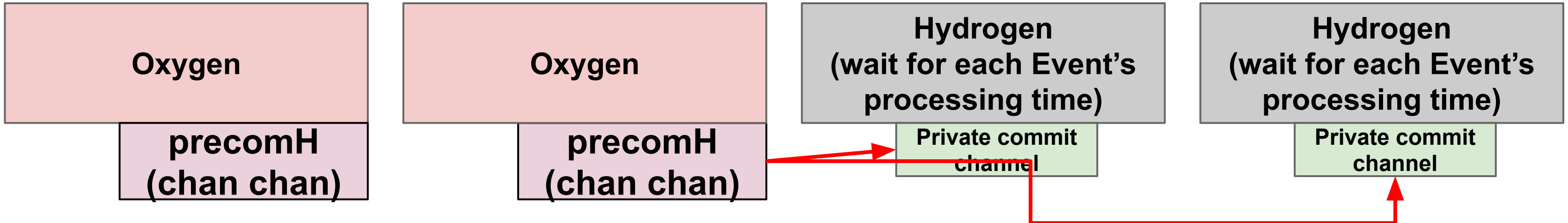

```go
// Step 4: Bond
bond()

// Step 5: (Postcommit)
//          Wait until the 2 hydrogen atoms to finish
// We re-use the same communication channel as (Commit)
<-h1
<-h2

// Step 6: Step down from being leader
wf.oxygenMutex <- struct{}{}
```

Why does this question matter?

- Regardless of the solution, the problem has the same **constraints**

- This shows how barriers, mutexes, channels, etc can help to enforce the **same constraints!**

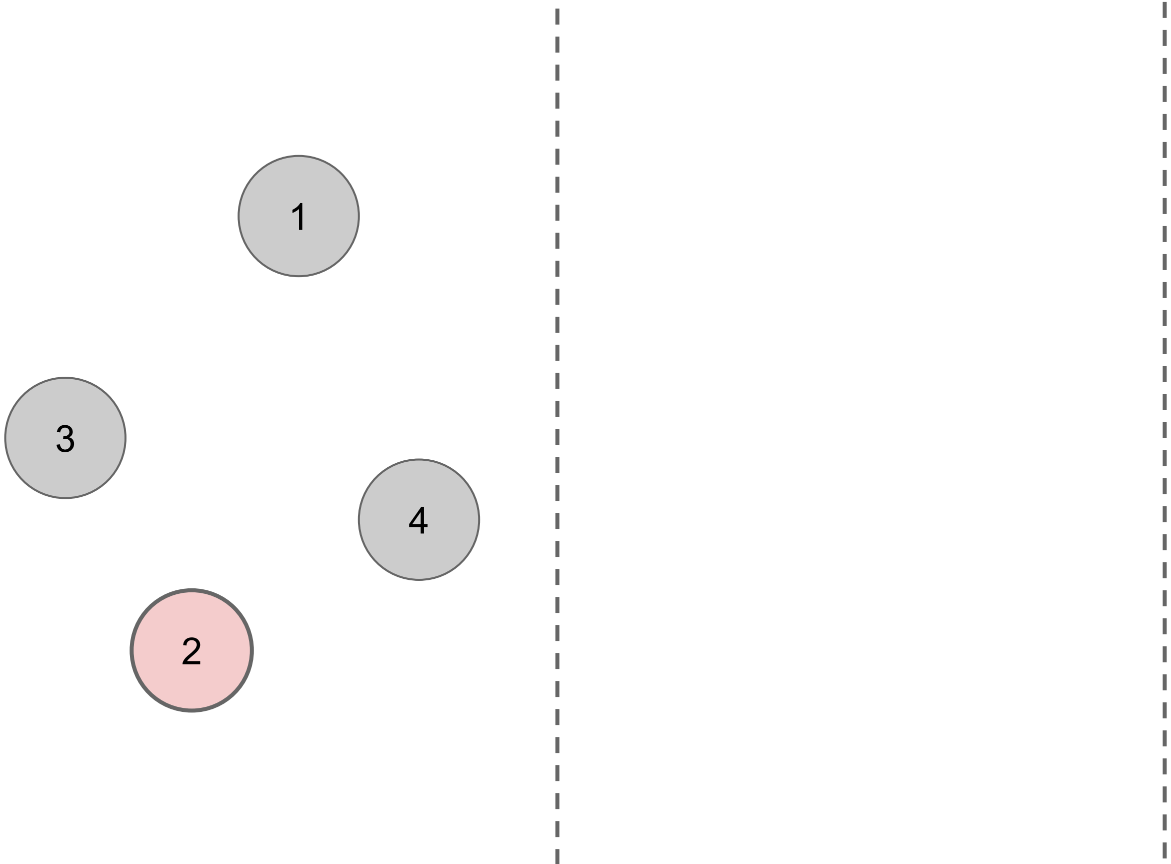




FIFO Semaphore

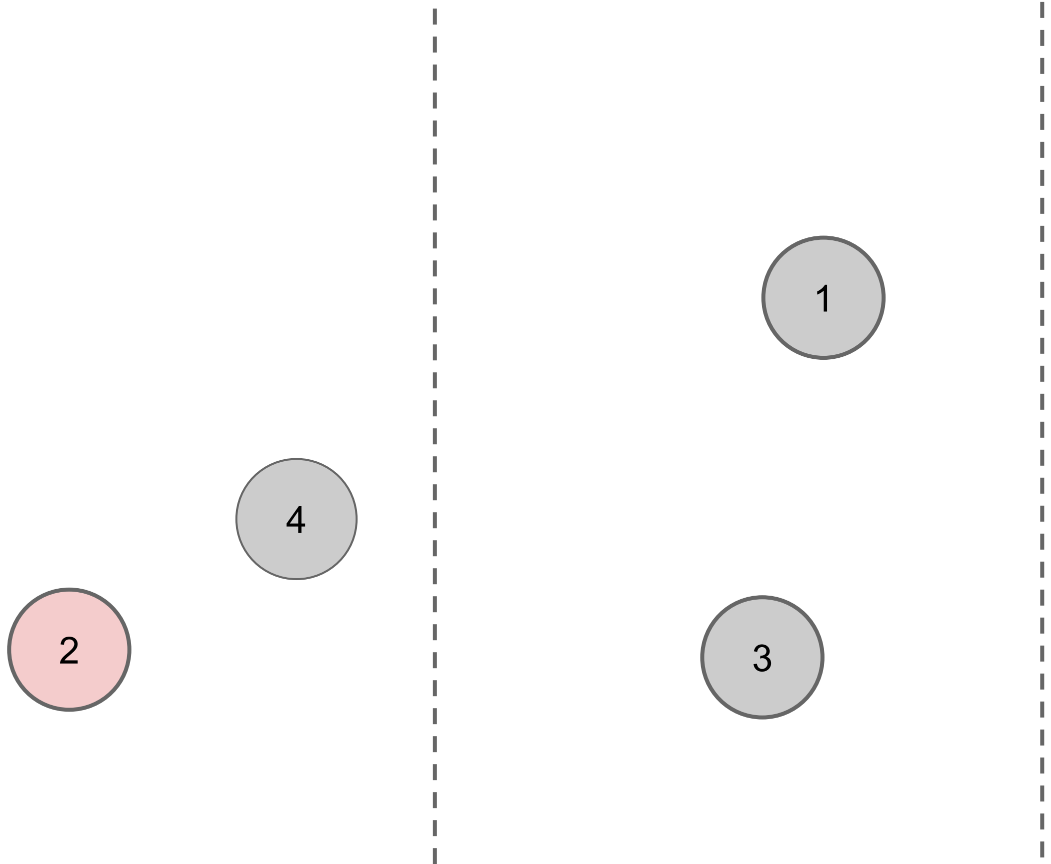
FIFO Semaphore Problem Statement

Example here is for capacity 2 semaphore



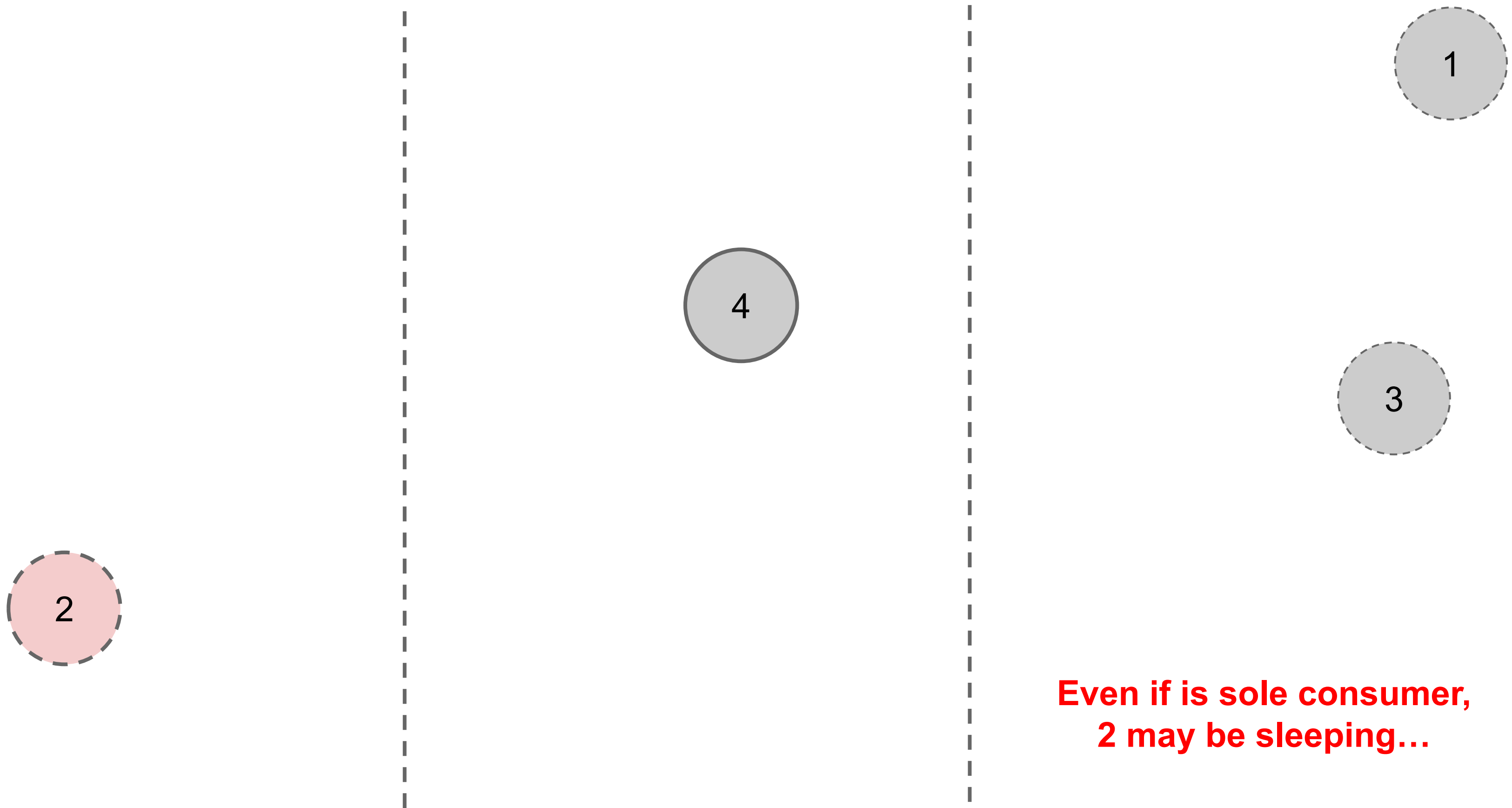
E.g. Oxygen atoms waiting to bond()

FIFO Semaphore Problem Statement



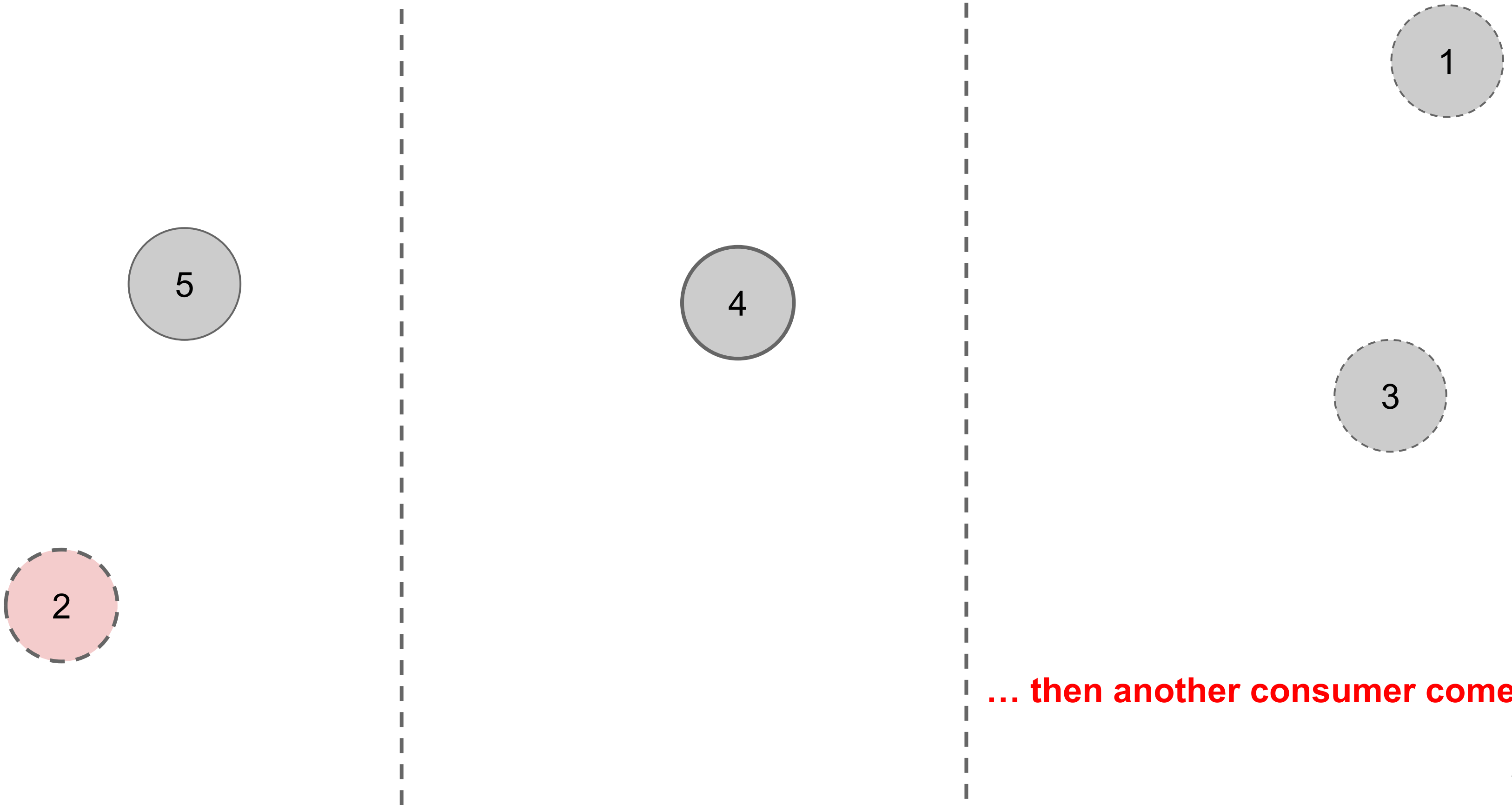
Semaphores don't guarantee order!

FIFO Semaphore Problem Statement



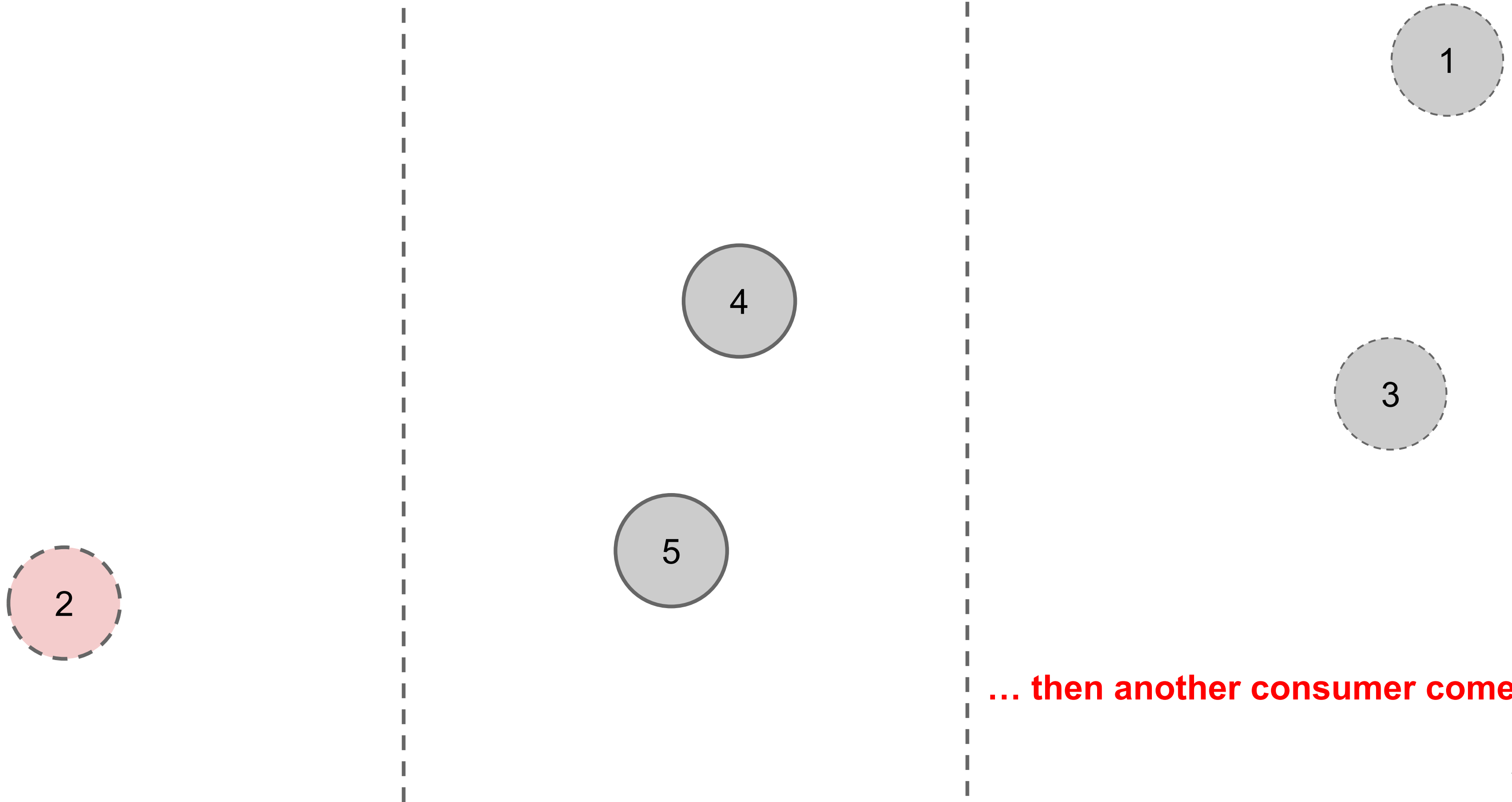
**Even if is sole consumer,
2 may be sleeping…**

FIFO Semaphore Problem Statement



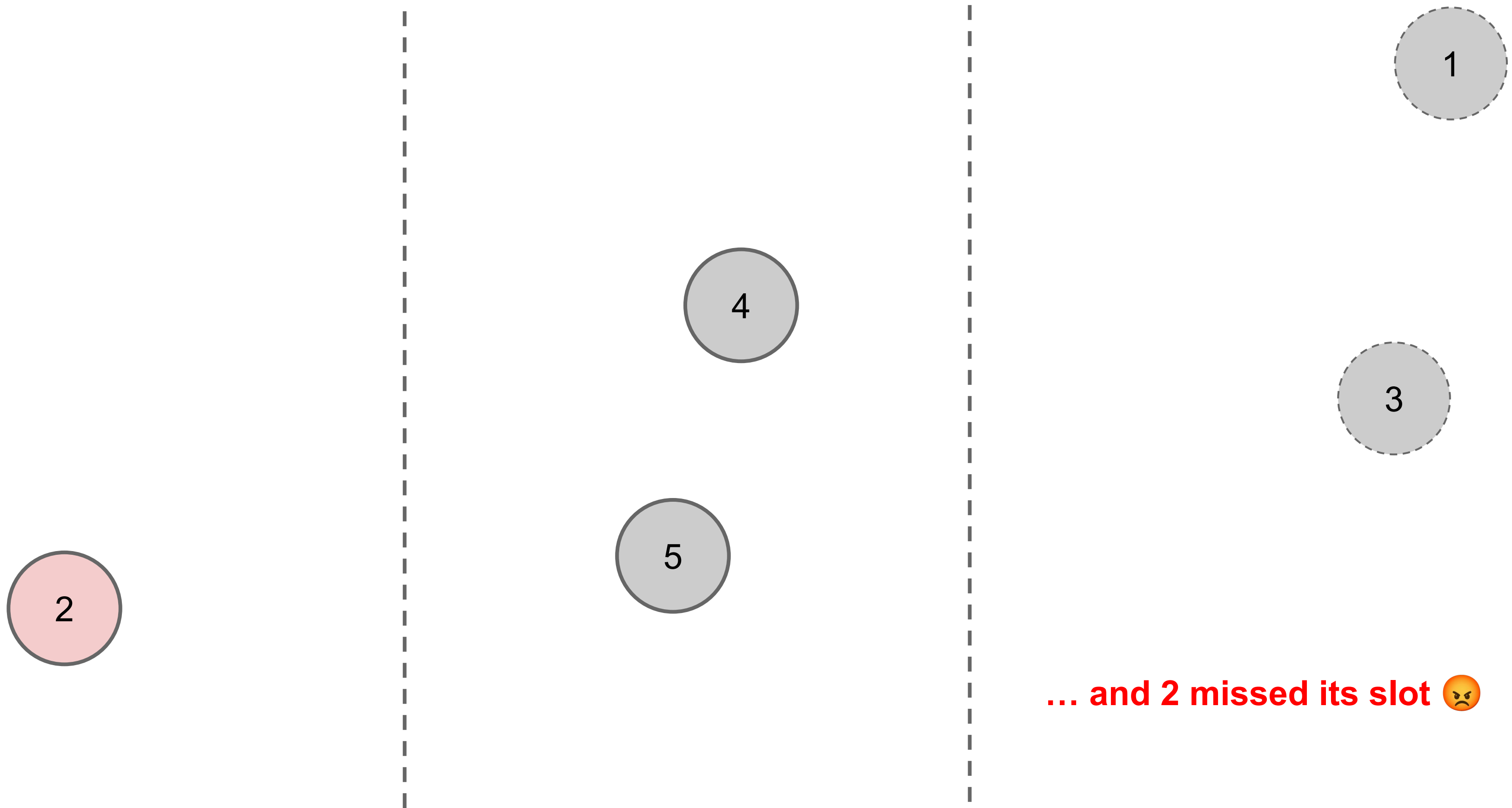
… then another consumer comes

FIFO Semaphore Problem Statement

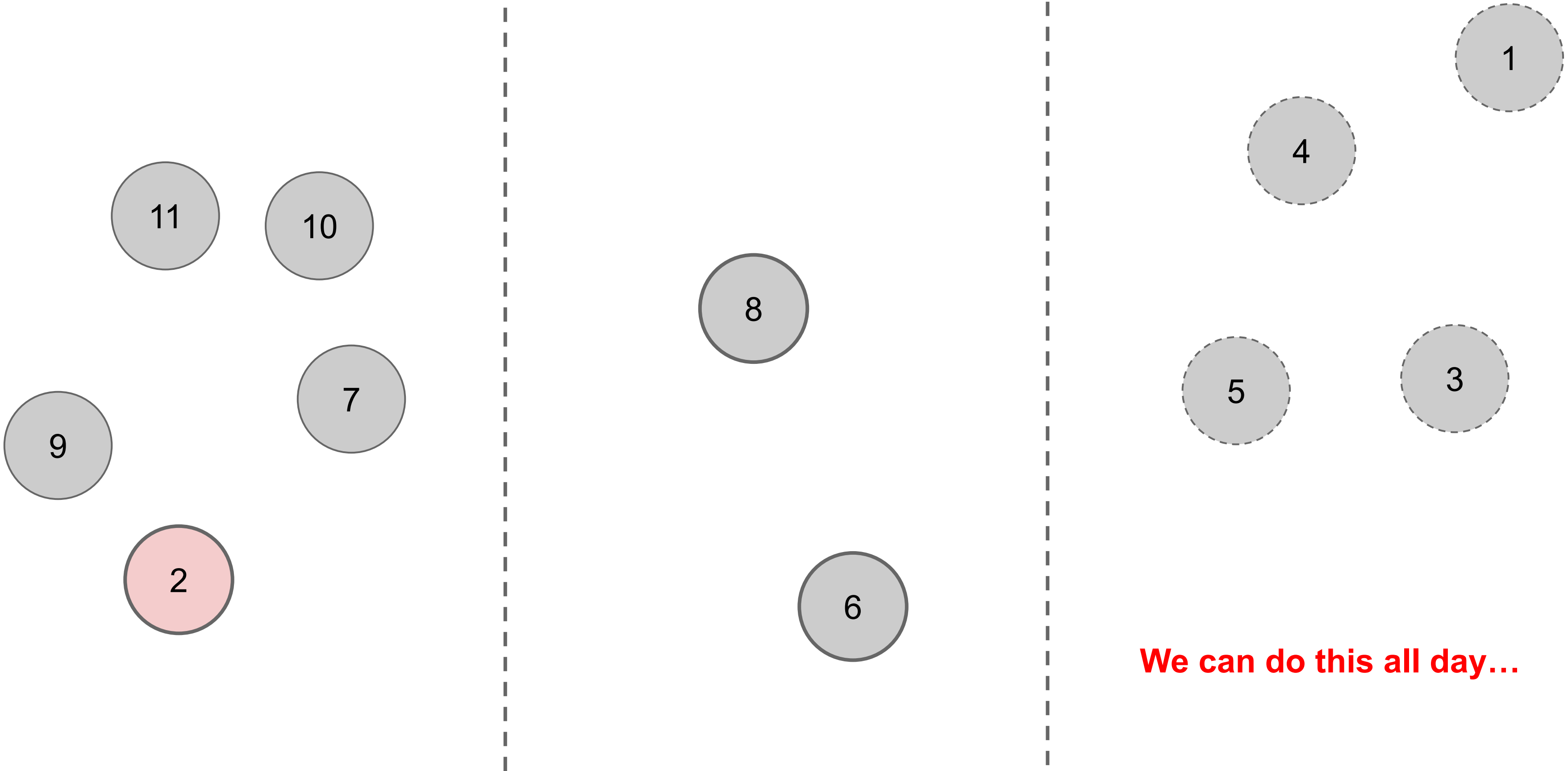


… then another consumer comes

FIFO Semaphore Problem Statement



FIFO Semaphore Problem Statement

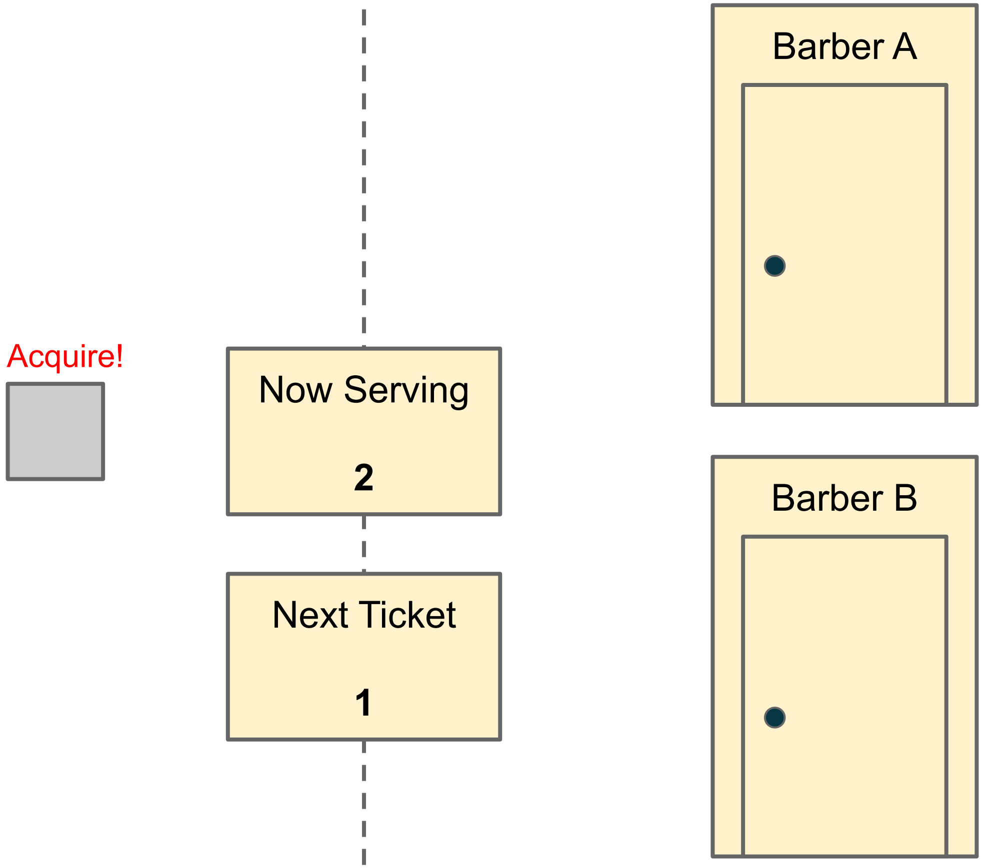


FIFO Semaphore:
Shared Memory Ticket Queue

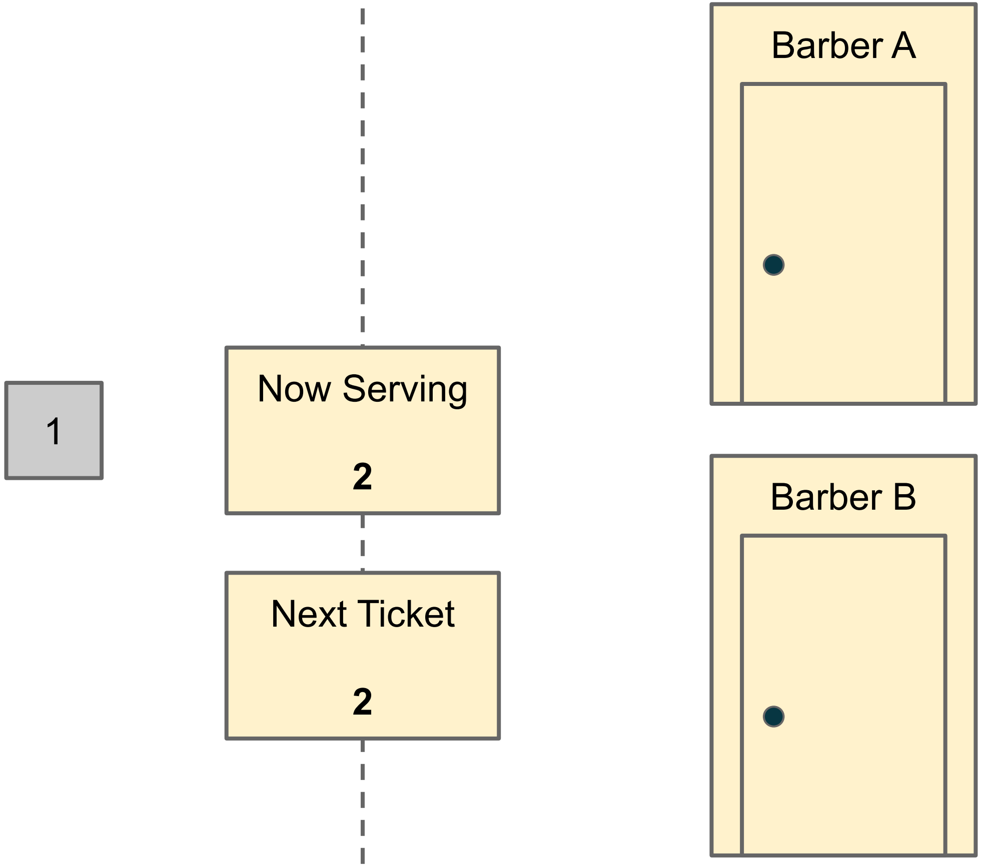
Demo 4: ticket queue

```
struct FIFOSemaphore {
  std::atomic<std::ptrdiff_t> next_ticket{1};
  std::atomic<std::ptrdiff_t> now_serving;

  FIFOSemaphore(std::ptrdiff_t initial_count) :
now_serving{initial_count} {}

  void acquire() {
    auto my_ticket = next_ticket.fetch_add(1);
    while (now_serving.load() < my_ticket) {}
  }

  void release() {
    now_serving.fetch_add(1);
  }
};
```

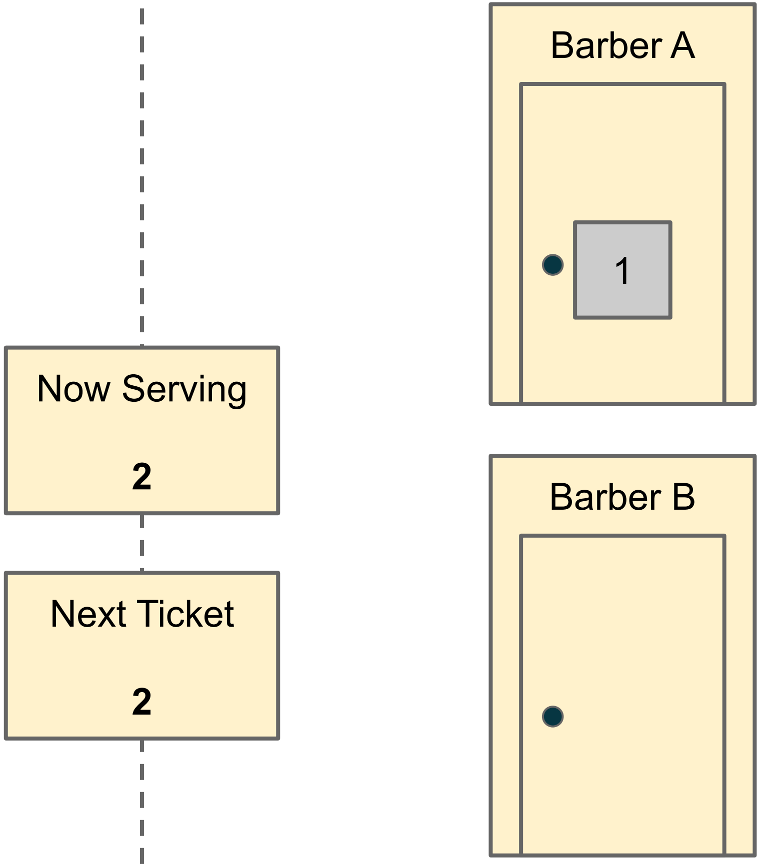
Demo 4: ticket queue (initial_count = 2)



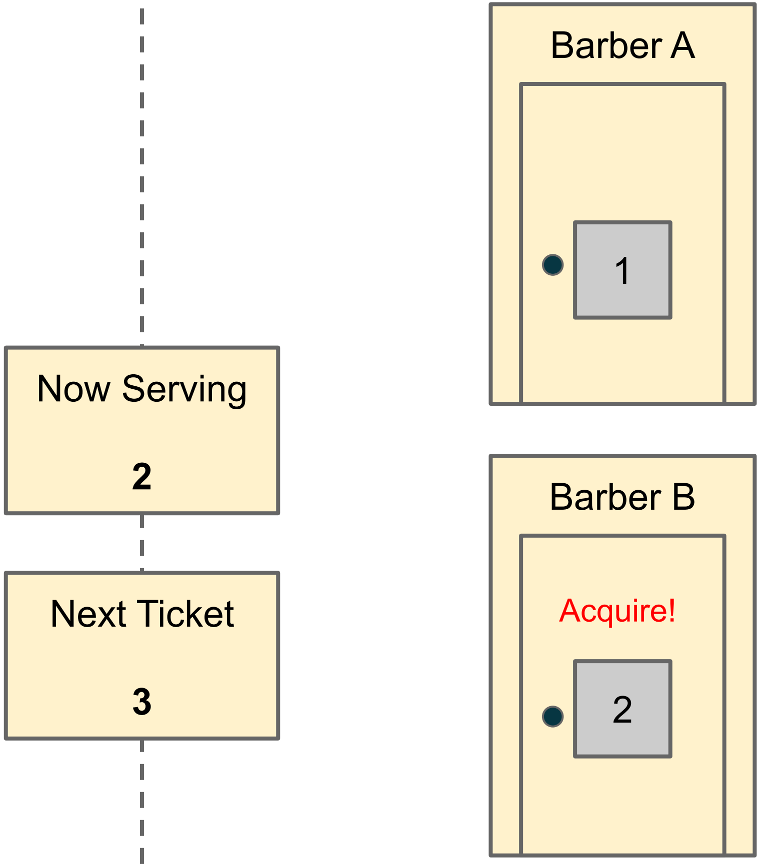
Demo 4: ticket queue (initial_count = 2)



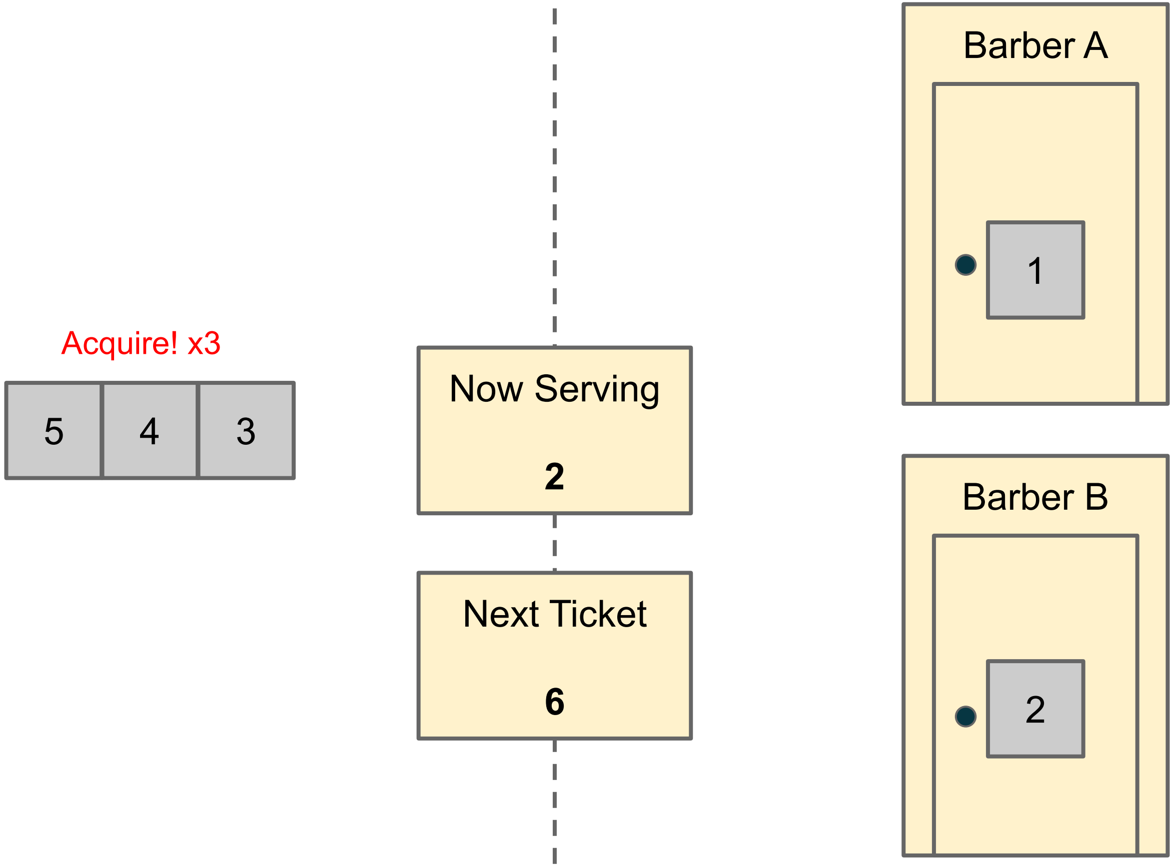
Demo 4: ticket queue (initial_count = 2)



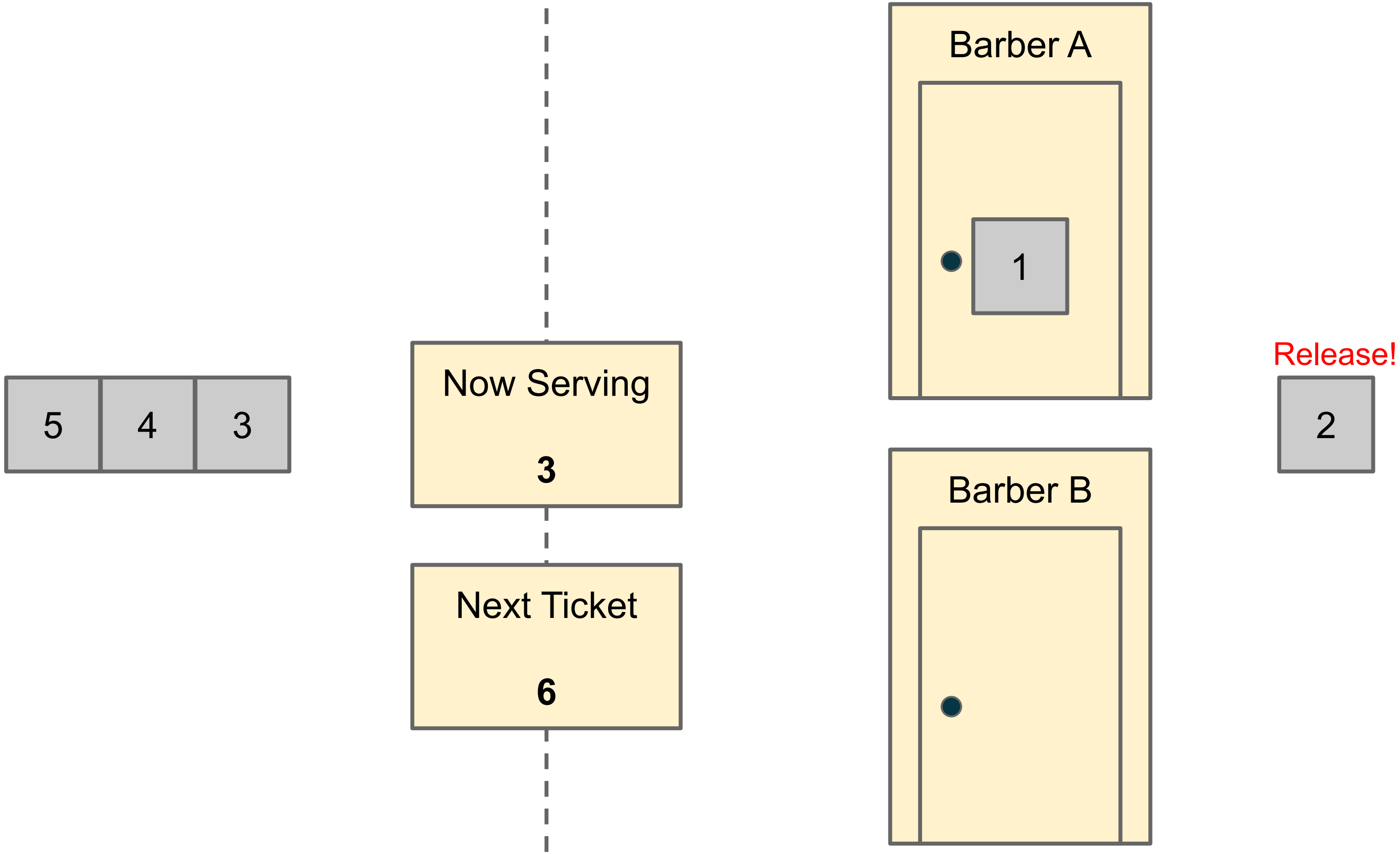
Demo 4: ticket queue (initial_count = 2)



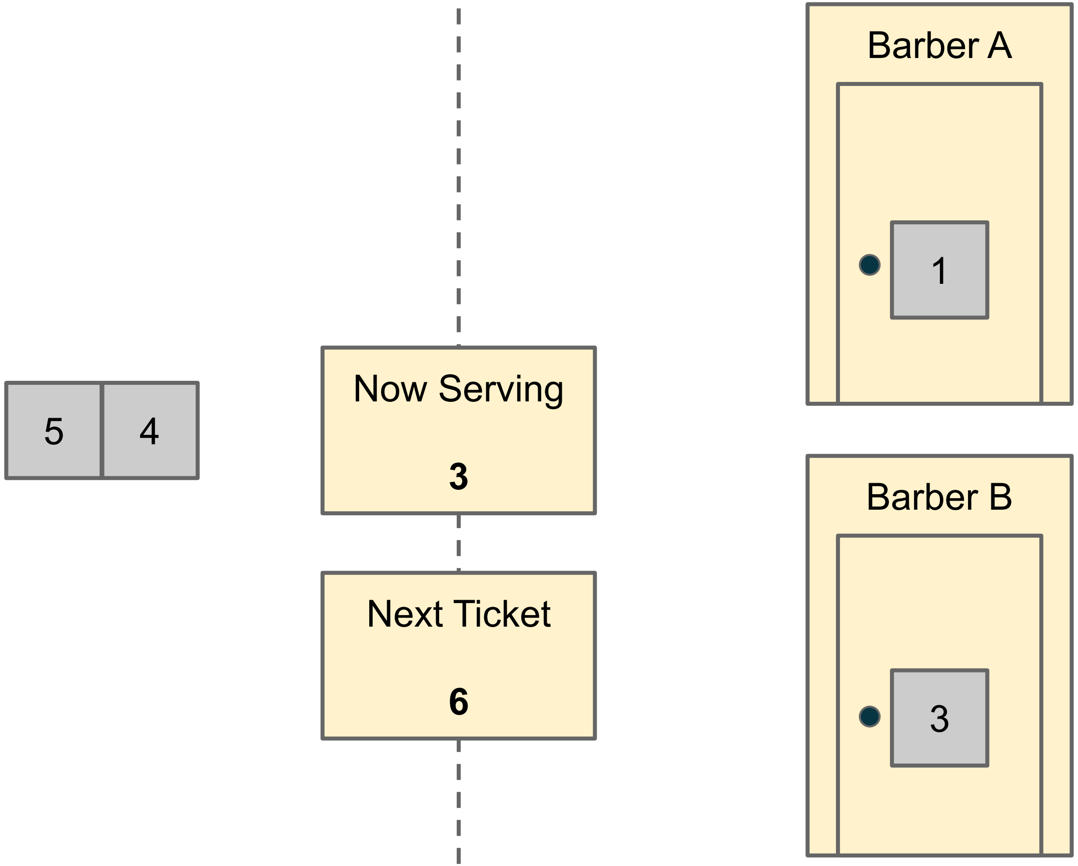
Demo 4: ticket queue (initial_count = 2)



Demo 4: ticket queue (initial_count = 2)



Demo 4: ticket queue (initial_count = 2)



Demo 4: ticket queue

```
struct FIFOSemaphore {
  std::atomic<std::ptrdiff_t> next_ticket{1};
  std::atomic<std::ptrdiff_t> now_serving;

  FIFOSemaphore(std::ptrdiff_t initial_count) :
now_serving{initial_count} {}

  void acquire() {
    auto my_ticket = next_ticket.fetch_add(1);
    while (now_serving.load() < my_ticket) {}
  }

  void release() {
    now_serving.fetch_add(1);
  }
};
```

How do we remove the busy-waiting from this example?

Demo 4: ticket queue: condvars

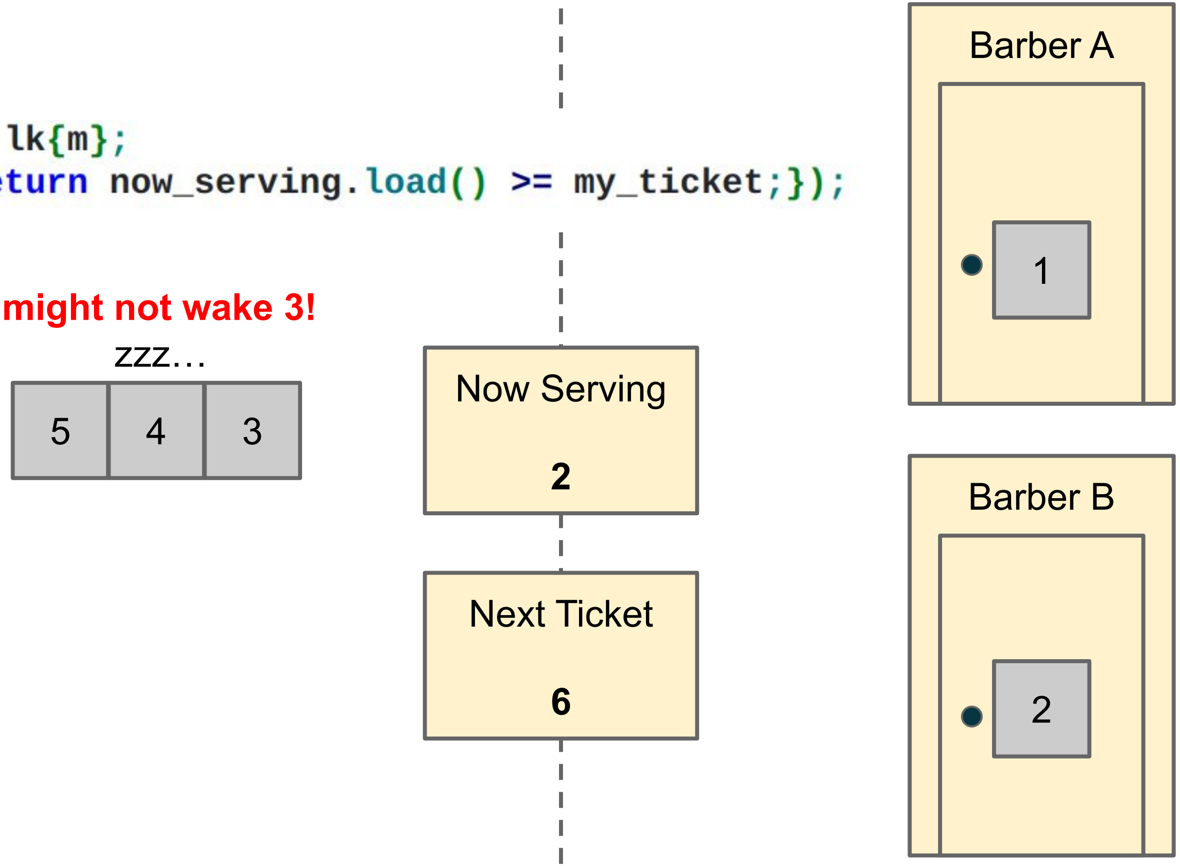
What's wrong with this solution?

```
struct FIFOSemaphore {
  ...
  std::atomic<uint> next_ticket{1}, now_serving;
  FIFOSemaphore(uint cap) : now_serving{cap} {}

  void acquire() {
    uint my_ticket = next_ticket.fetch_add(1);
    std::unique_lock lk{m};
    cv.wait(lk, []{return now_serving.load() >= my_ticket;});
  }

  void release() {
    now_serving.fetch_add(1);
    cv.notify_one();
  }
};
```

Task 1: Replace spinwait

```
std::unique_lock lk{m};
cv.wait(lk, []{return now_serving.load() >= my_ticket;});
```

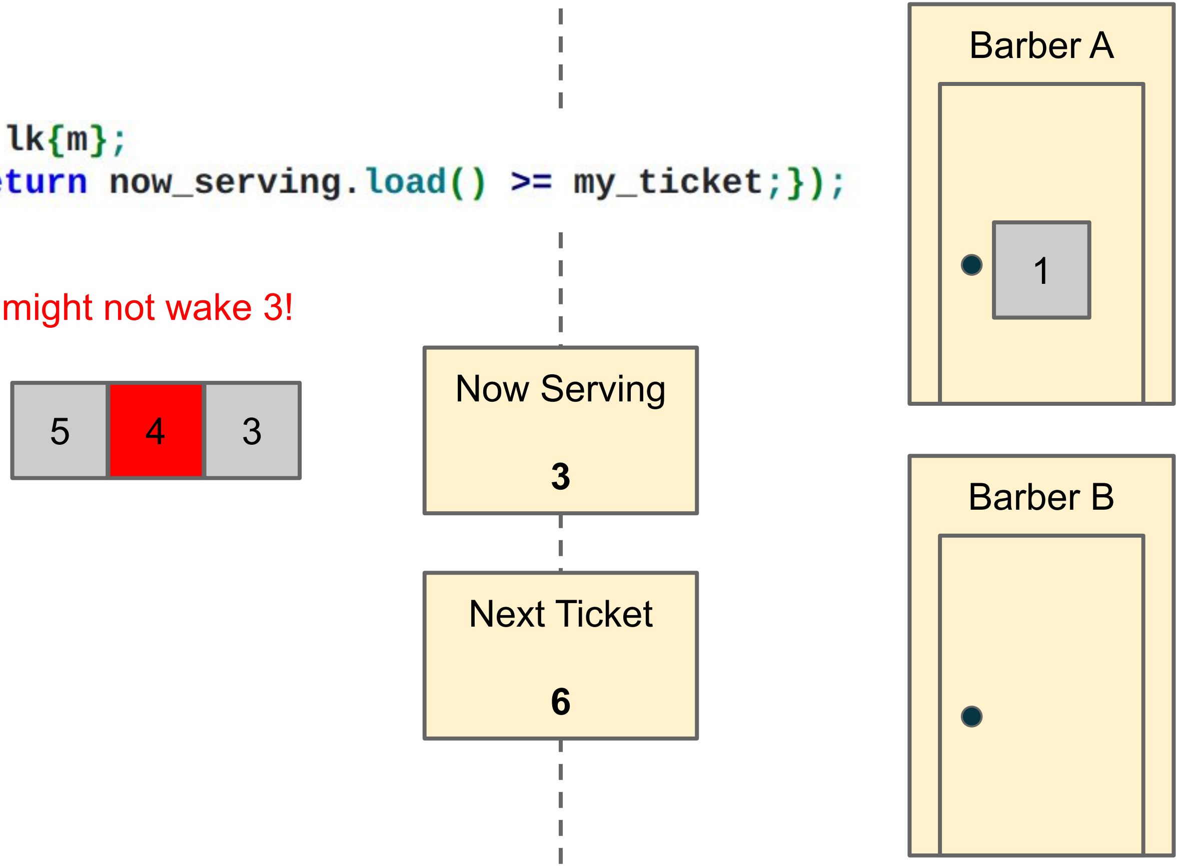
`notify_one()` **might not wake 3!**



Task 1: Replace spinwait

```
std::unique_lock lk{m};
cv.wait(lk, []{return now_serving.load() >= my_ticket;});
```

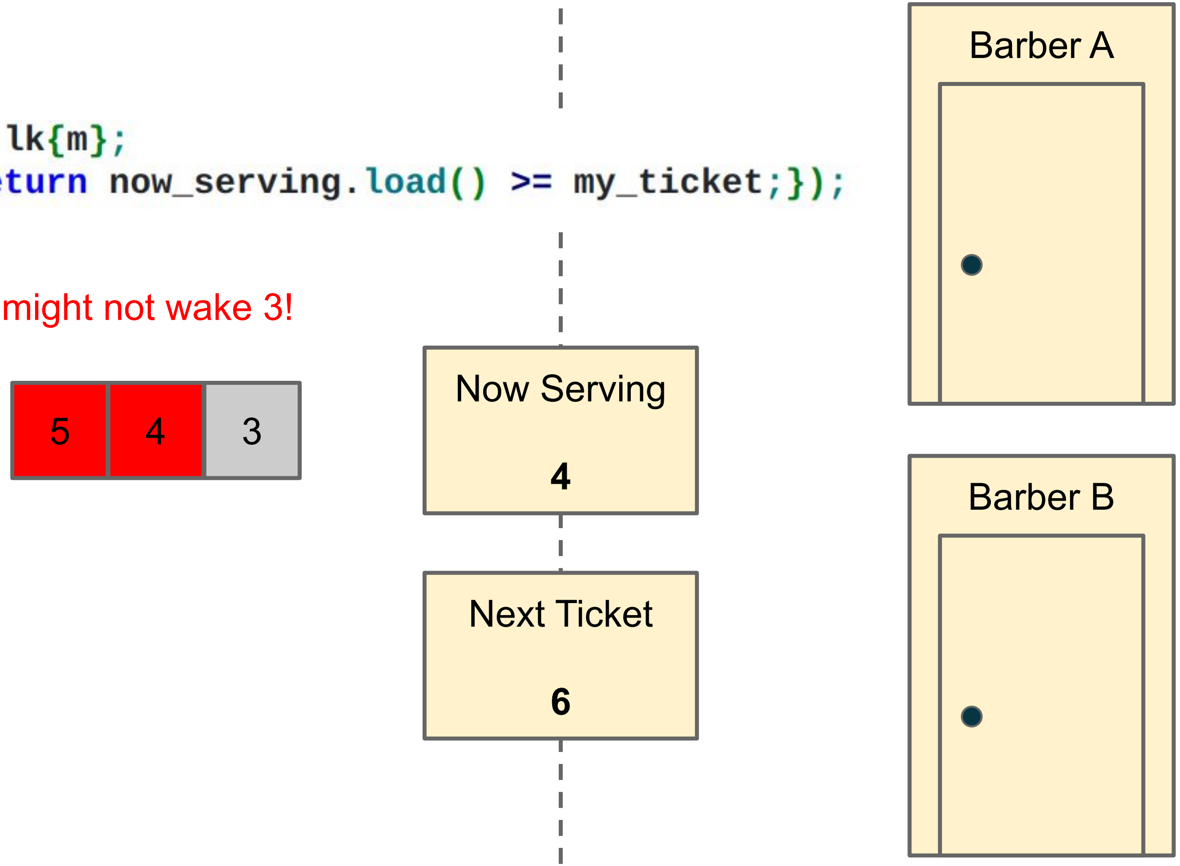
`notify_one()` might not wake 3!



Task 1: Replace spinwait

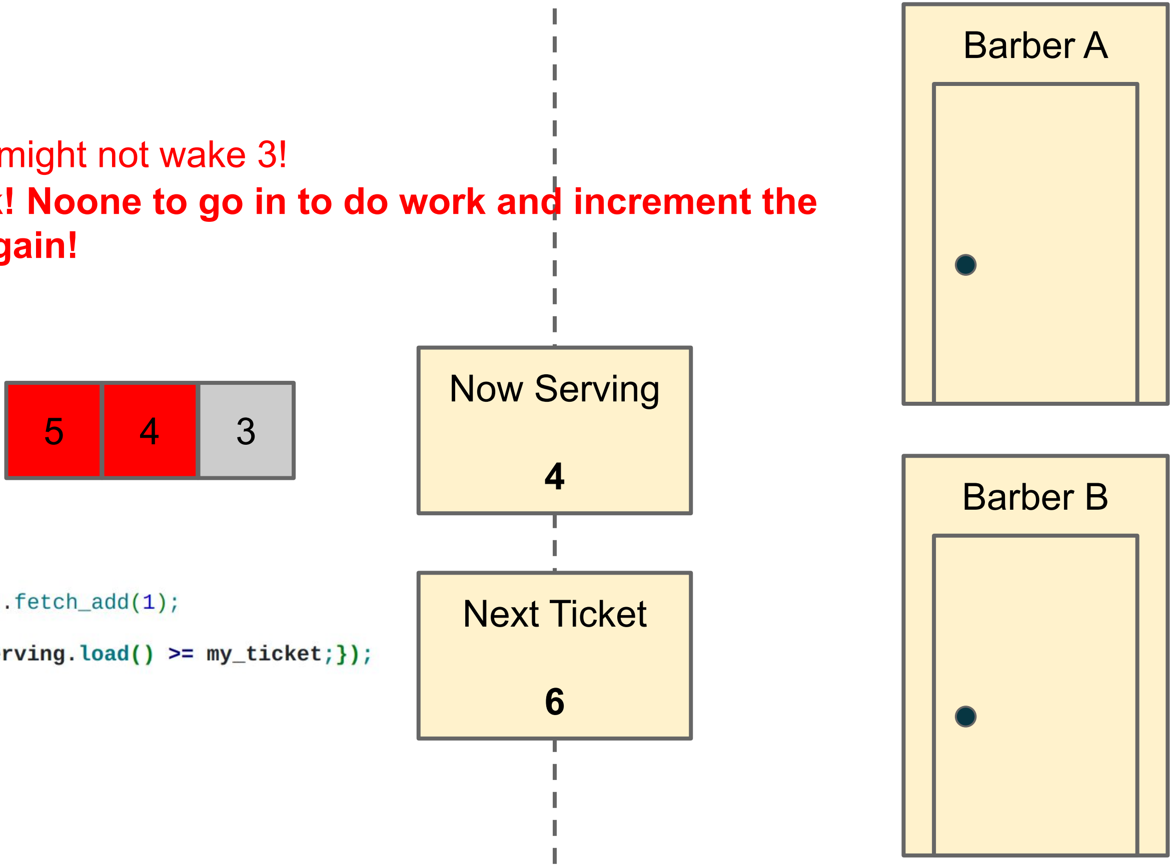
```
std::unique_lock lk{m};
cv.wait(lk, []{return now_serving.load() >= my_ticket;});
```

`notify_one()` might not wake 3!



Task 1: Replace spinwait

`notify_one()` might not wake 3!
Possible deadlock! Noone to go in to do work and increment the counter + notify again!




```
void acquire() {
  uint my_ticket = next_ticket.fetch_add(1);
  std::unique_lock lk{m};
  cv.wait(lk, []{return now_serving.load() >= my_ticket;});
}

void release() {
  now_serving.fetch_add(1);
  cv.notify_one();
}
```

Task 1: Replace spinwait

```
struct FIFOSemaphore {
  ...
  std::atomic<uint> next_ticket{1}, now_serving;
  FIFOSemaphore(uint cap) : now_serving{cap} {}

  void acquire() {
    uint my_ticket = next_ticket.fetch_add(1);
    std::unique_lock lk{m};
    cv.wait(lk, []{return now_serving.load() >= my_ticket;});
  }

  void release() {
    now_serving.fetch_add(1);
    cv.notify_all();
  }
};
```

Notifying all solves the problem, even though it makes threads wake up spuriously

FIFO Semaphore: Semaphore Queue

Demo 5: Queue of semaphores

```
void acquire() {
  auto waiter = std::make_shared<Waiter>();
  {
    std::scoped_lock lock{mut};
    if (count > 0) {
      count--;  // Positive count,
      return;   // simply decrement without blocking
    }
    waiters.push(waiter);  // Zero count, add to waiters
  }
  waiter->sem.acquire();  // and block on the semaphore
}
```

```
void release() {
  std::shared_ptr<Waiter> waiter;
  {
    std::scoped_lock lock{mut};
    if (waiters.empty()) {
      count++;  // No waiters, simply increment count
      return;
    }

    waiter = waiters.front();  // Pop a waiter
    waiters.pop();
  }
  waiter->sem.release();  // and signal it
}
```

```
struct FIFOSemaphore5 {
  struct Waiter {
    std::binary_semaphore sem{0};
  };

  std::mutex mut;
  std::queue<std::shared_ptr<Waiter>> waiters;
  std::ptrdiff_t count;

  FIFOSemaphore5(std::ptrdiff_t initial_count)
      : mut{}, waiters{}, count{initial_count} {}
```

Why are we blocking on binary semaphores?

Why not a mutex?

Demo 5: Queue of semaphores

```
void acquire() {
  auto waiter = std::make_shared<Waiter>();
  {
    std::scoped_lock lock{mut};
    if (count > 0) {
      count--;  // Positive count,
      return;   // simply decrement without blocking
    }
    waiters.
  }
  waiter->se
}
```

```
void rel
  std::s
  {
    std:
    if (
      co
      return;
    }

    waiter = waiters.front();  // Pop a waiter
    waiters.pop();
  }
  waiter->sem.release();  // and signal it
}
```

```
struct FIFOSemaphore5 {
  struct Waiter {
    std::binary_semaphore sem{0};
  };

  std::mutex mut;
  std::queue<std::shared_ptr<Waiter>> waiters;
```

Why are we blocking on binary semaphores?

Why not a mutex?

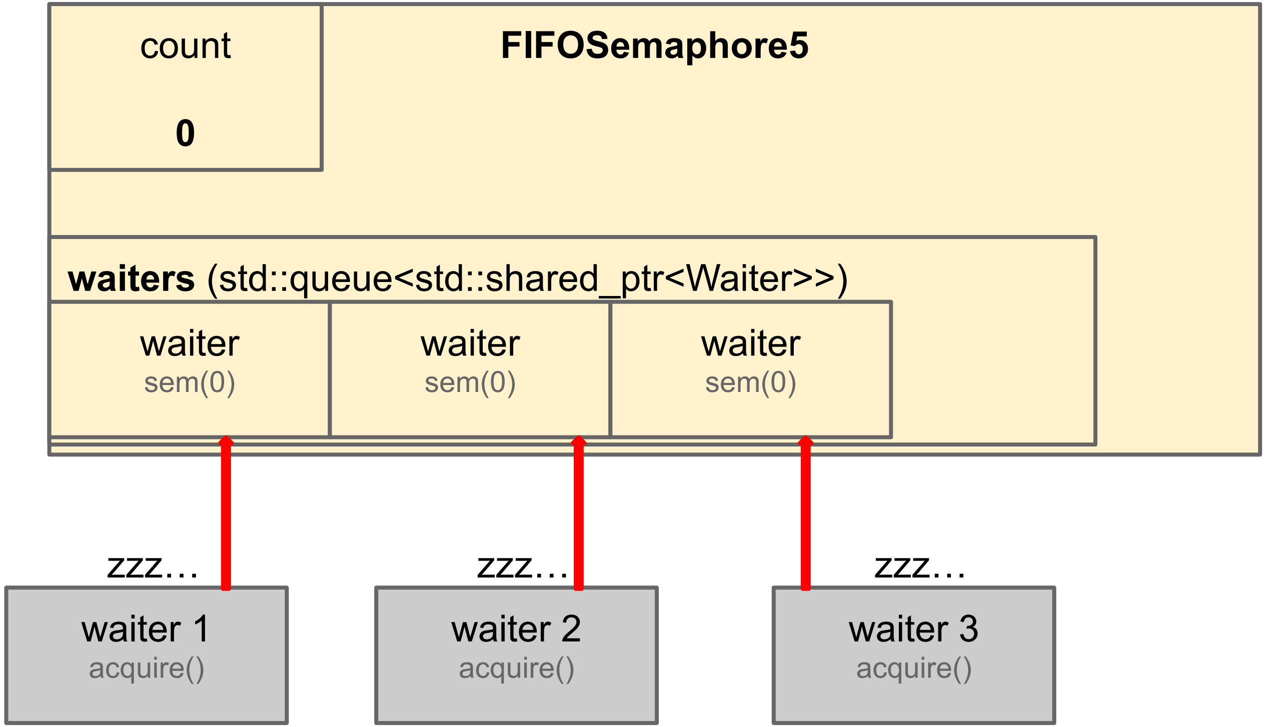
std::mutex::unlock

```
void unlock();
```
(since C++11)

Unlocks the mutex.

The mutex must be locked by the current thread of execution, otherwise, the behavior is undefined.

Demo 5: Queue of semaphores



Demo 5: Queue of semaphores

FIFOSemaphore5

count

0

waiters (std::queue<std::shared_ptr<Waiter>>)

waiter
sem(0)

waiter
sem(0)

Release!!

waiter
sem(**1**)

waiter 1
acquire()

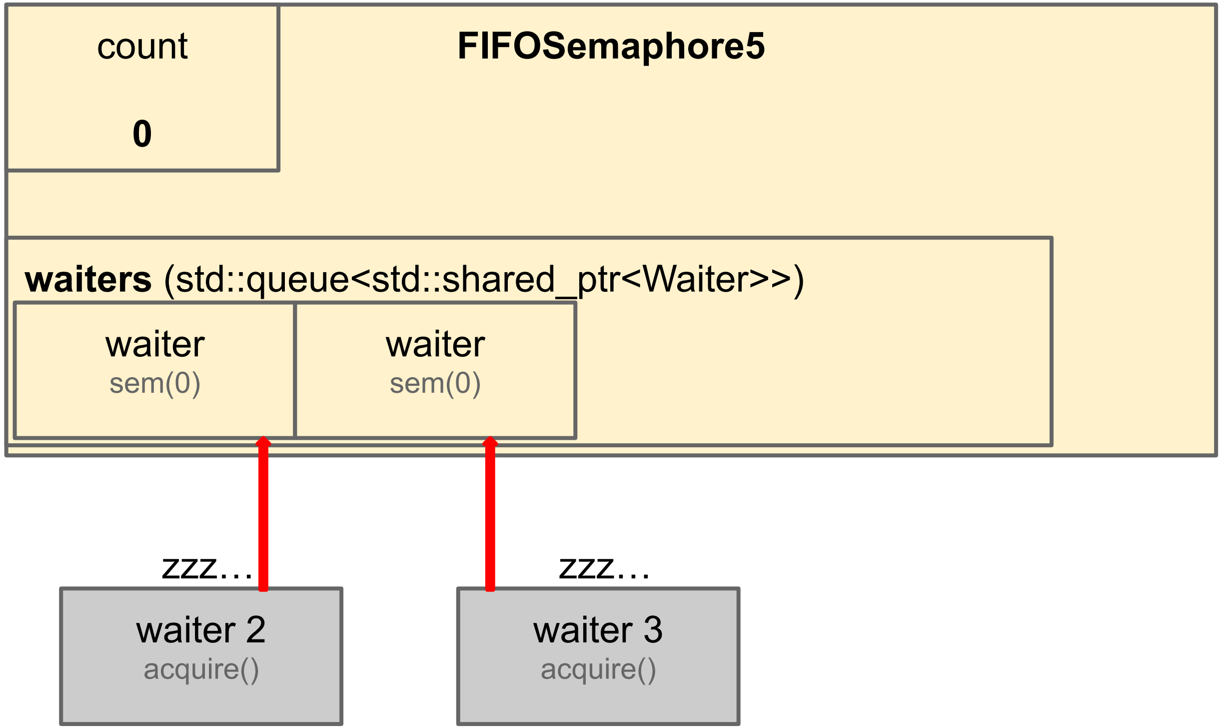
zzz…

waiter 2
acquire()

zzz…

waiter 3
acquire()

Demo 5: Queue of semaphores



Demo 5: Queue of semaphores

FIFOSemaphore5

count

0

waiters (std::queue<std::shared_ptr<Waiter>>)

waiter
sem(0)

waiter
sem(0)

zzz…

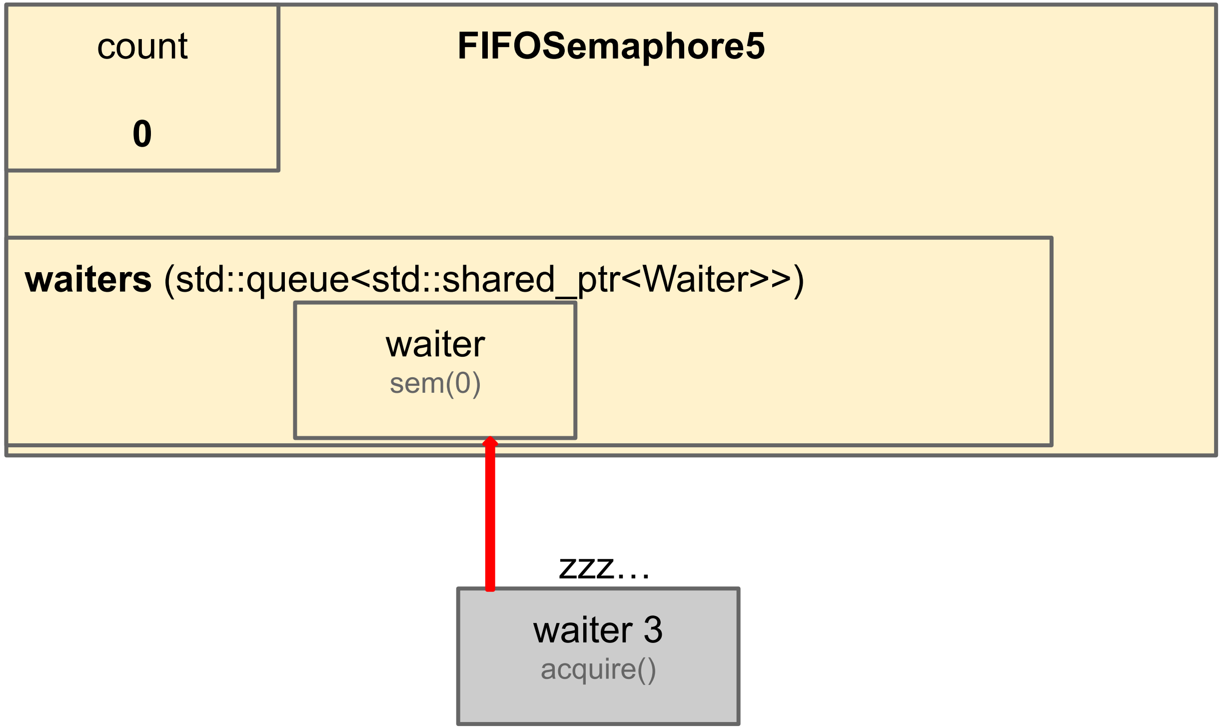
waiter 2
acquire()

zzz…

waiter 3
acquire()

Release!!

Demo 5: Queue of semaphores



Demo 5: Queue of semaphores

count

1

FIFOSemaphore5

waiters (std::queue<std::shared_ptr<Waiter>>)

waiter
sem(0)

zzz…

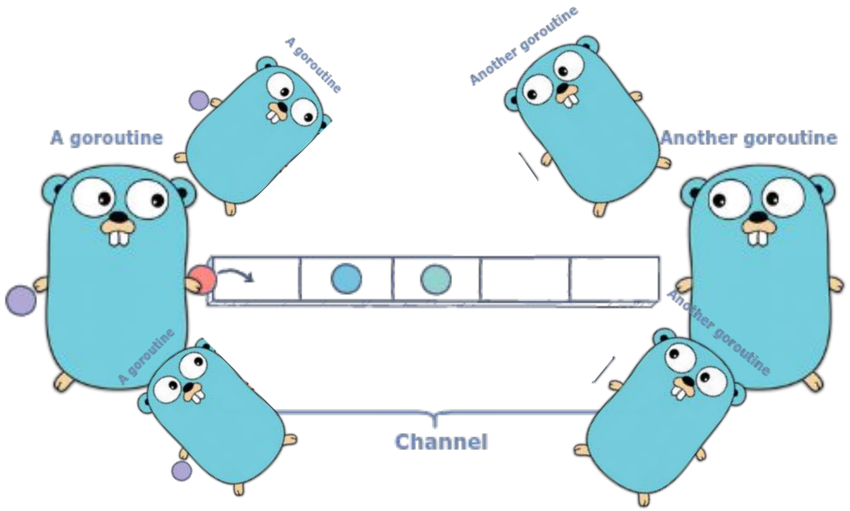
waiter 3
acquire()

Release!!

FIFO Semaphore: Go

Buffered channel == FIFO semaphore?

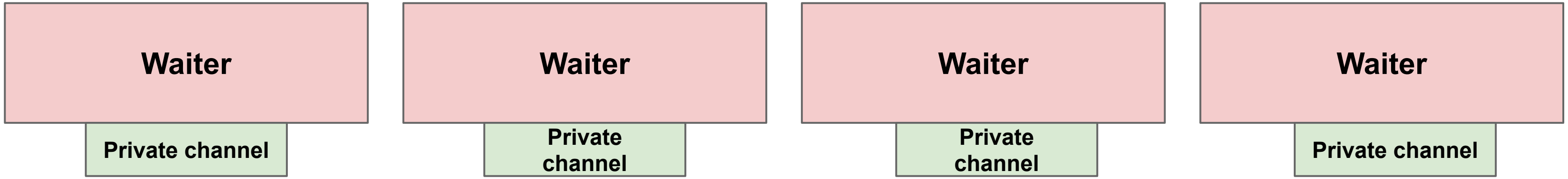
- Not guaranteed by the Go standard!

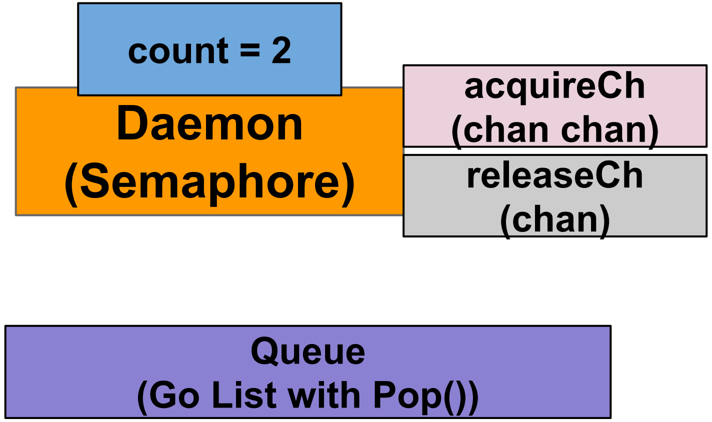


FIFO Semaphore: Go: Daemon Solution

Go: FIFO Daemon Solution

- Daemon waits for someone to acquire



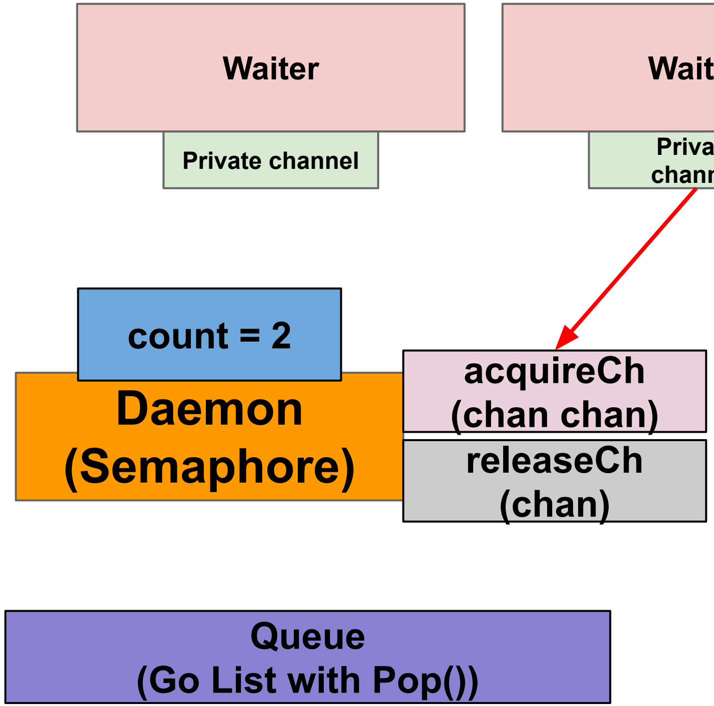



```go
case ch := <-sem.acquireCh: // Decrement or add a waiter
    if count > 0 {
        count--
        ch <- struct{}{} // Don't keep waiter blocked
    } else {
        waiters.PushBack(ch) // Add waiter to back of queue
    }
}
```

Go: FIFO Daemon Solution

- Daemon waits for someone to acquire

```go
func (s *Semaphore2) Acquire() {
    ch := make(chan struct{})
    // Send daemon a channel that can be used to unblock us
    s.acquireCh <- ch
    // Block until daemon decides to unblock us
    <-ch
}
```

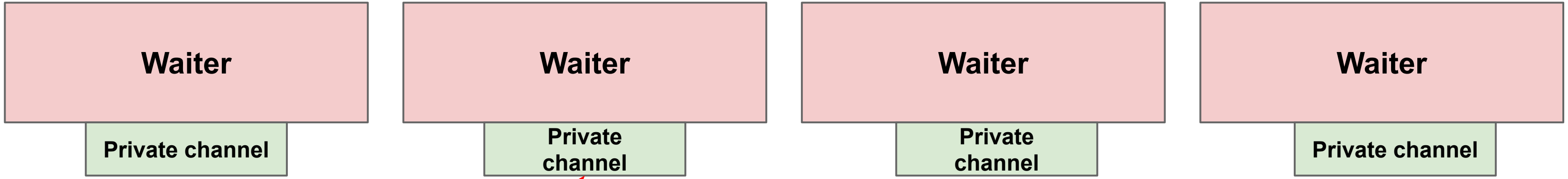


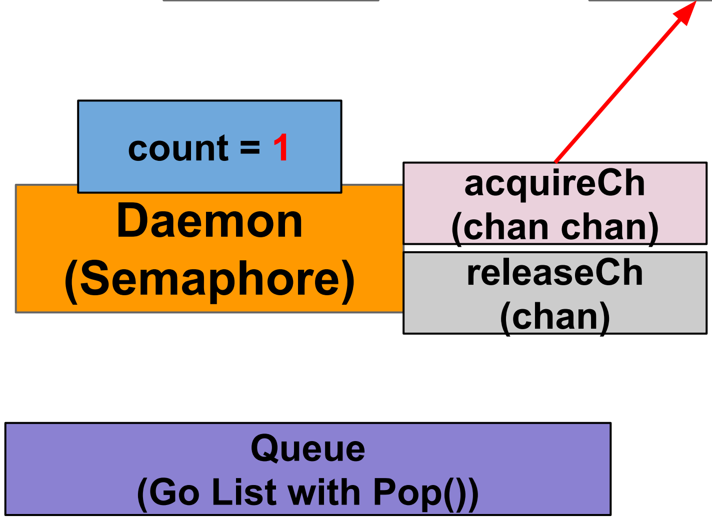

```go
case ch := <-sem.acquireCh: // Decrement or add a waiter
    if count > 0 {
        count--
        ch <- struct{}{} // Don't keep waiter blocked
    } else {
        waiters.PushBack(ch) // Add waiter to back of queue
    }
}
```

Go: FIFO Daemon Solution

```go
func (s *Semaphore2) Acquire() {
    ch := make(chan struct{})
    // Send daemon a channel that can be used to unblock us
    s.acquireCh <- ch
    // Block until daemon decides to unblock us
    <-ch
}
```

- Count is decremented since > 0
- Waiter is sent on its way



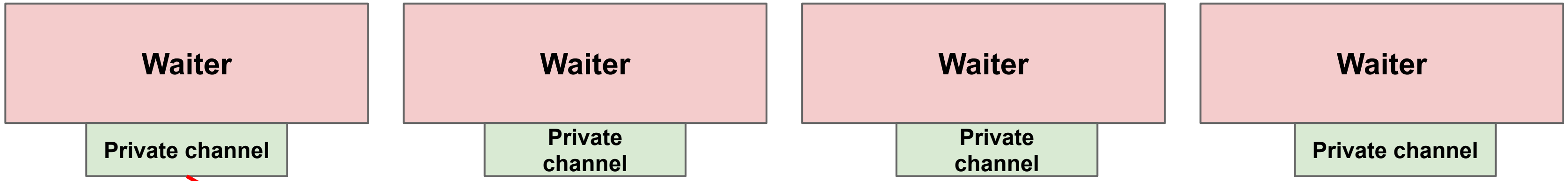


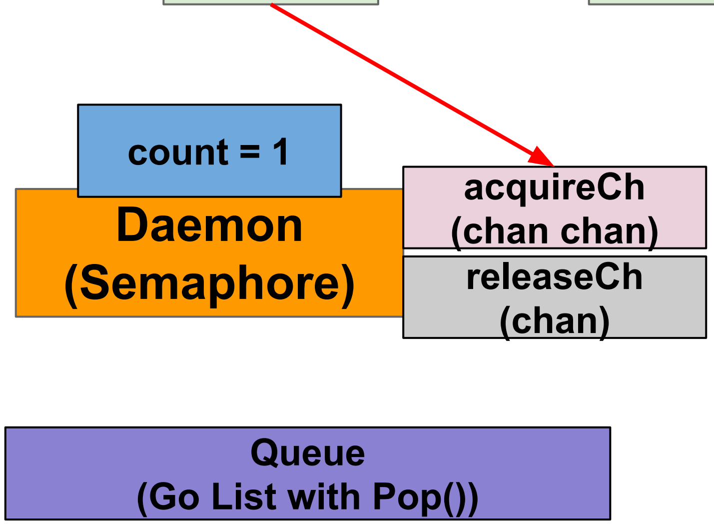

```go
case ch := <-sem.acquireCh: // Decrement or add a waiter
    if count > 0 {
        count--
        ch <- struct{}{} // Don't keep waiter blocked
    } else {
        waiters.PushBack(ch) // Add waiter to back of queue
    }
}
```

Go: FIFO Daemon Solution

```go
func (s *Semaphore2) Acquire() {
    ch := make(chan struct{})
    // Send daemon a channel that can be used to unblock us
    s.acquireCh <- ch
    // Block until daemon decides to unblock us
    <-ch
}
```

- Daemon waits for someone to acquire



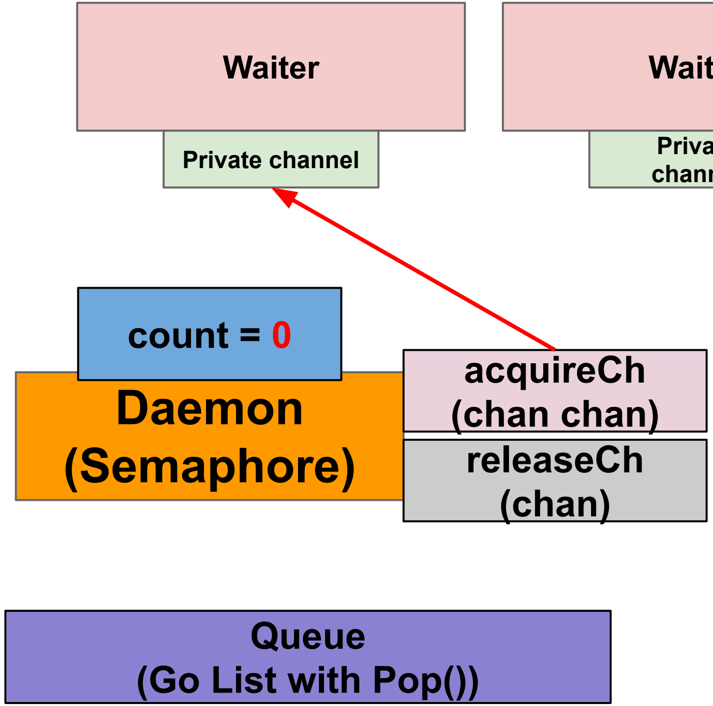



```go
case ch := <-sem.acquireCh: // Decrement or add a waiter
    if count > 0 {
        count--
        ch <- struct{}{} // Don't keep waiter blocked
    } else {
        waiters.PushBack(ch) // Add waiter to back of queue
    }
}
```

Go: FIFO Daemon Solution

```go
func (s *Semaphore2) Acquire() {
    ch := make(chan struct{})
    // Send daemon a channel that can be used to unblock us
    s.acquireCh <- ch
    // Block until daemon decides to unblock us
    <-ch
}
```

- Count is decremented since > 0
- Waiter is sent on its way

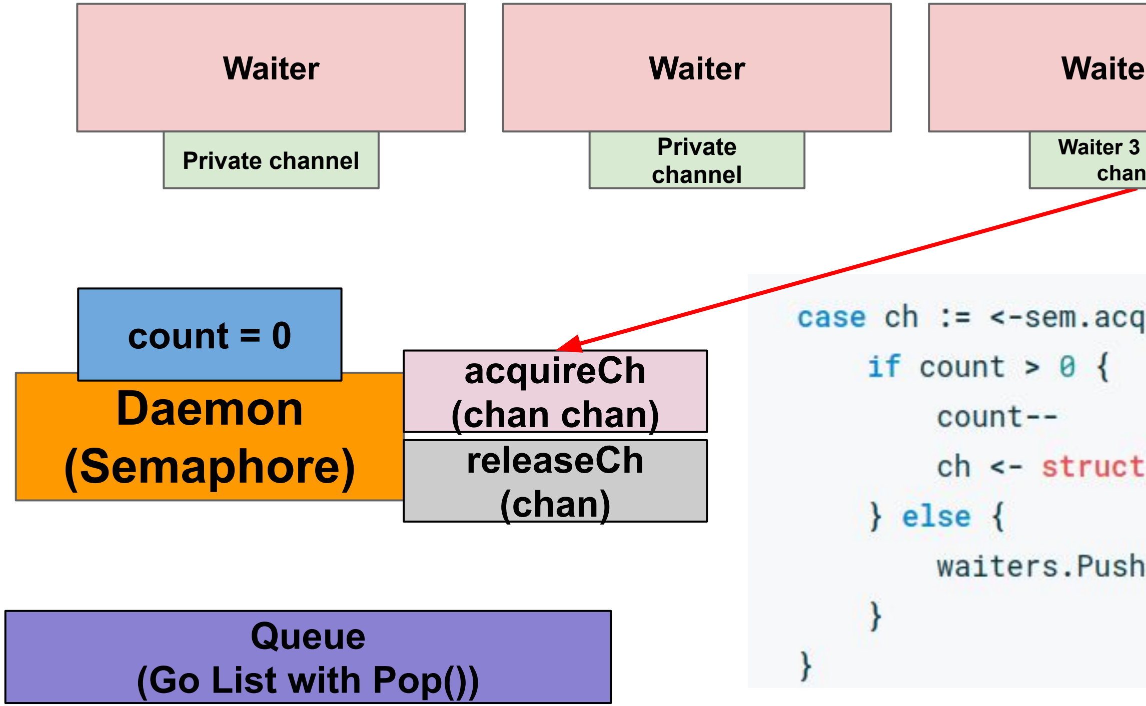



```go
case ch := <-sem.acquireCh: // Decrement or add a waiter
    if count > 0 {
        count--
        ch <- struct{}{} // Don't keep waiter blocked
    } else {
        waiters.PushBack(ch) // Add waiter to back of queue
    }
}
```

Go: FIFO Daemon Solution

- Daemon waits for someone to acquire

```go
func (s *Semaphore2) Acquire() {
    ch := make(chan struct{})
    // Send daemon a channel that can be used to unblock us
    s.acquireCh <- ch
    // Block until daemon decides to unblock us
    <-ch
}
```

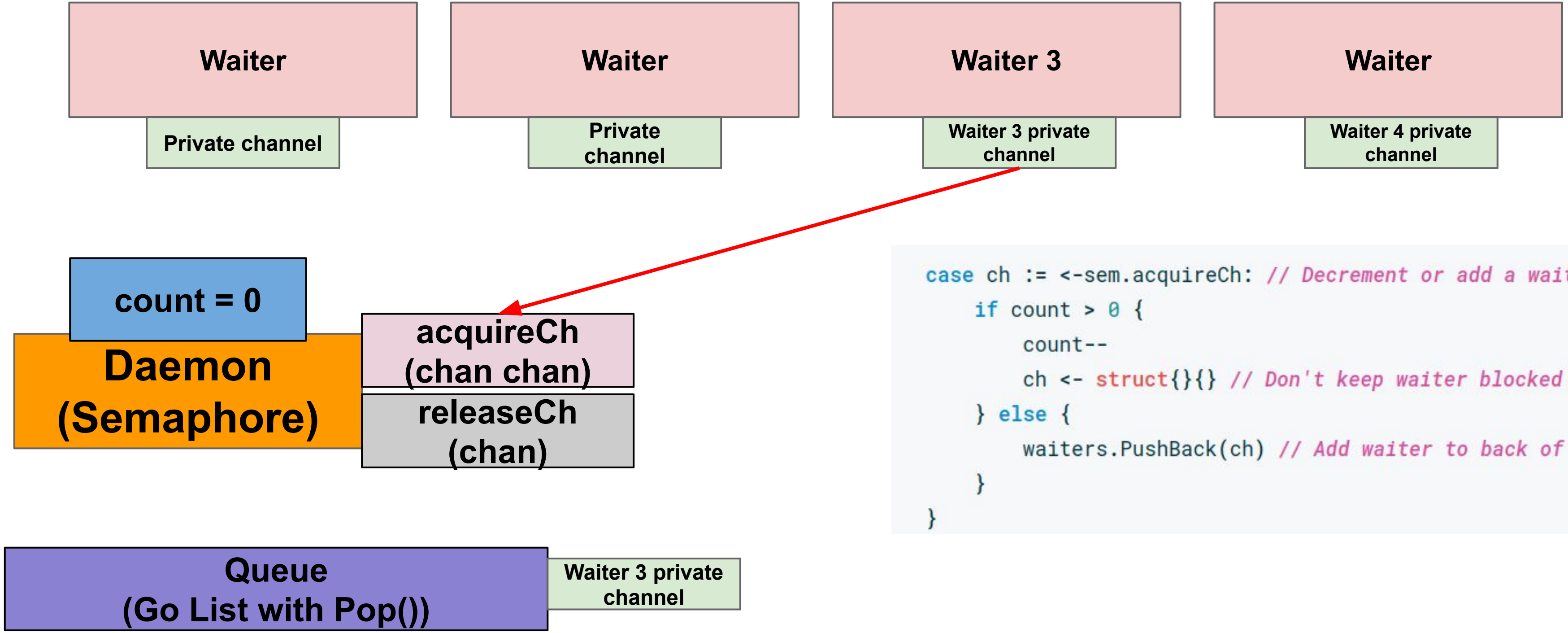



```go
case ch := <-sem.acquireCh: // Decrement or add a waiter
    if count > 0 {
        count--
        ch <- struct{}{} // Don't keep waiter blocked
    } else {
        waiters.PushBack(ch) // Add waiter to back of queue
    }
}
```

Go: FIFO Daemon Solution

```go
func (s *Semaphore2) Acquire() {
    ch := make(chan struct{})
    // Send daemon a channel that can be used to unblock us
    s.acquireCh <- ch
    // Block until daemon decides to unblock us
    <-ch
}
```

- Count is 0, so waiter is blocked
- Waiter’s private channel is pushed into the queue

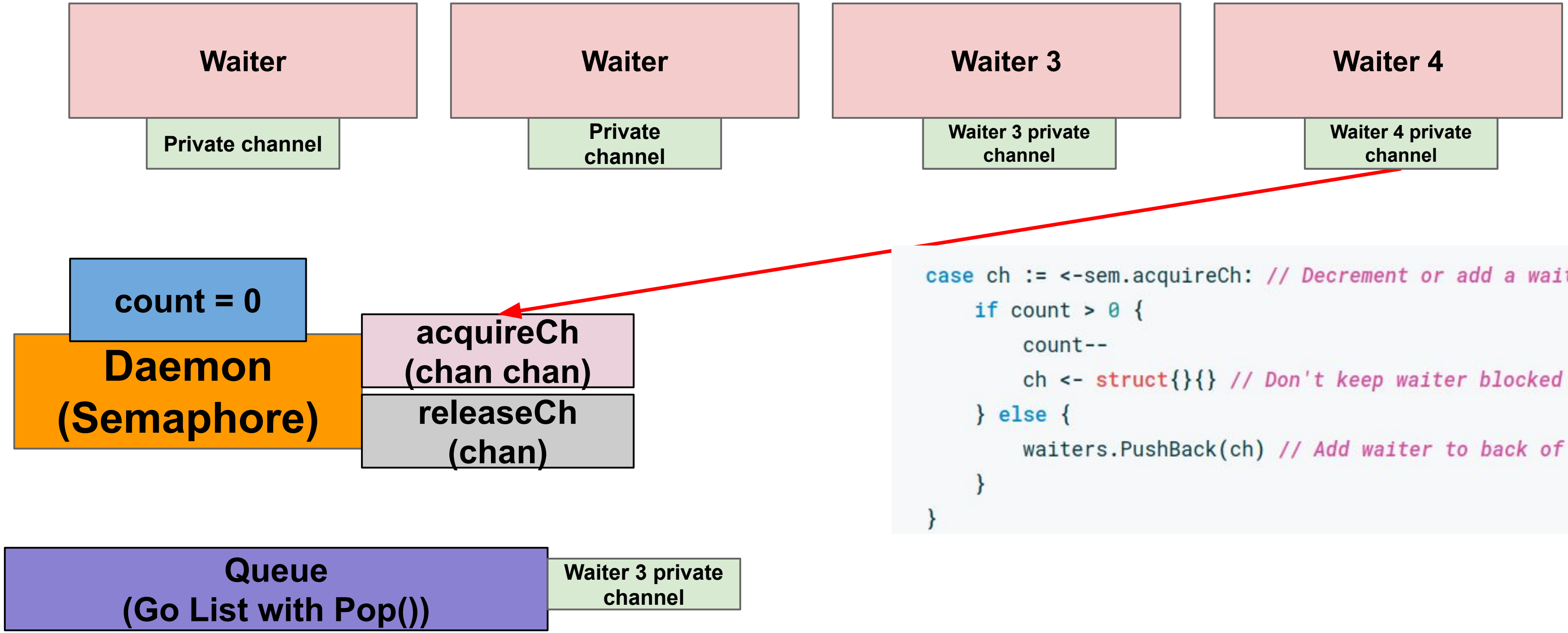



```go
case ch := <-sem.acquireCh: // Decrement or add a waiter
    if count > 0 {
        count--
        ch <- struct{}{} // Don't keep waiter blocked
    } else {
        waiters.PushBack(ch) // Add waiter to back of queue
    }
}
```

Go: FIFO Daemon Solution

- Daemon waits for someone to acquire

```go
func (s *Semaphore2) Acquire() {
    ch := make(chan struct{})
    // Send daemon a channel that can be used to unblock us
    s.acquireCh <- ch
    // Block until daemon decides to unblock us
    <-ch
}
```

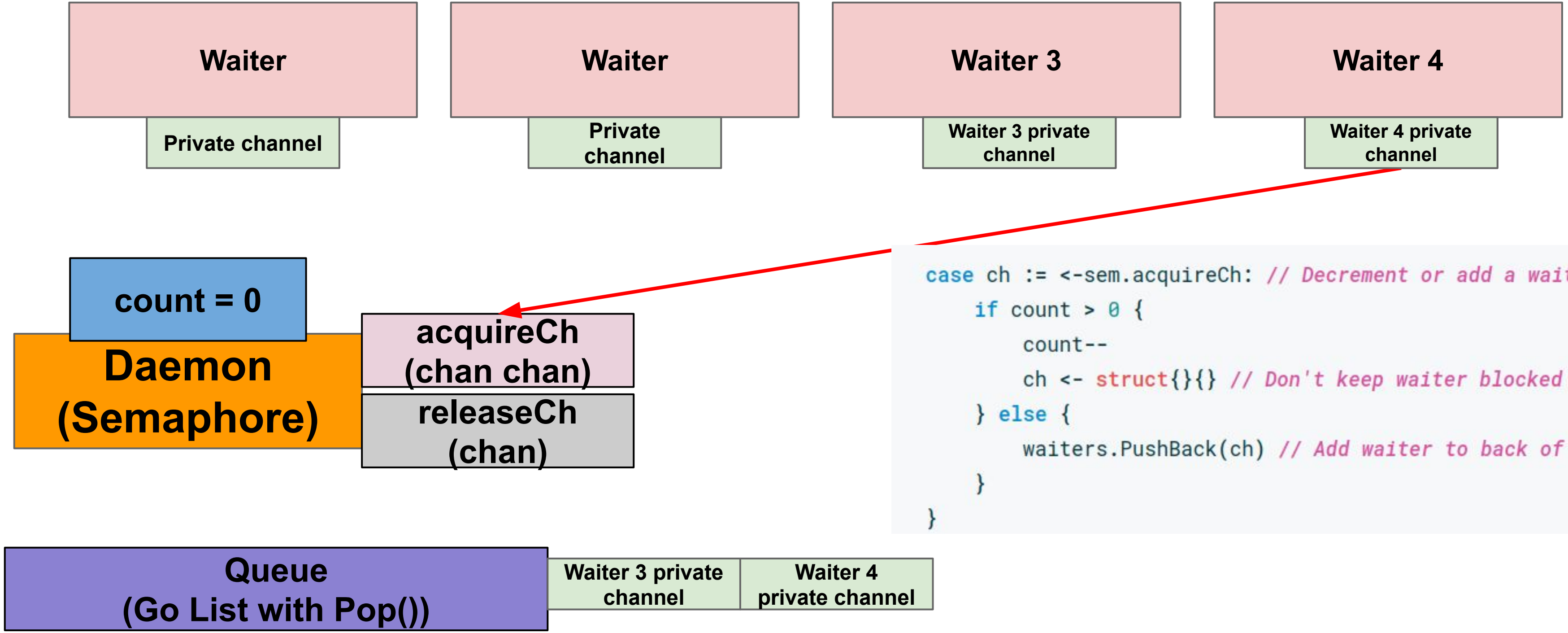



```go
case ch := <-sem.acquireCh: // Decrement or add a waiter
    if count > 0 {
        count--
        ch <- struct{}{} // Don't keep waiter blocked
    } else {
        waiters.PushBack(ch) // Add waiter to back of queue
    }
}
```

Go: FIFO Daemon Solution

- Count is 0, so waiter is blocked
- Waiter's private channel is pushed into the queue

```go
func (s *Semaphore2) Acquire() {
    ch := make(chan struct{})
    // Send daemon a channel that can be used to unblock us
    s.acquireCh <- ch
    // Block until daemon decides to unblock us
    <-ch
}
```

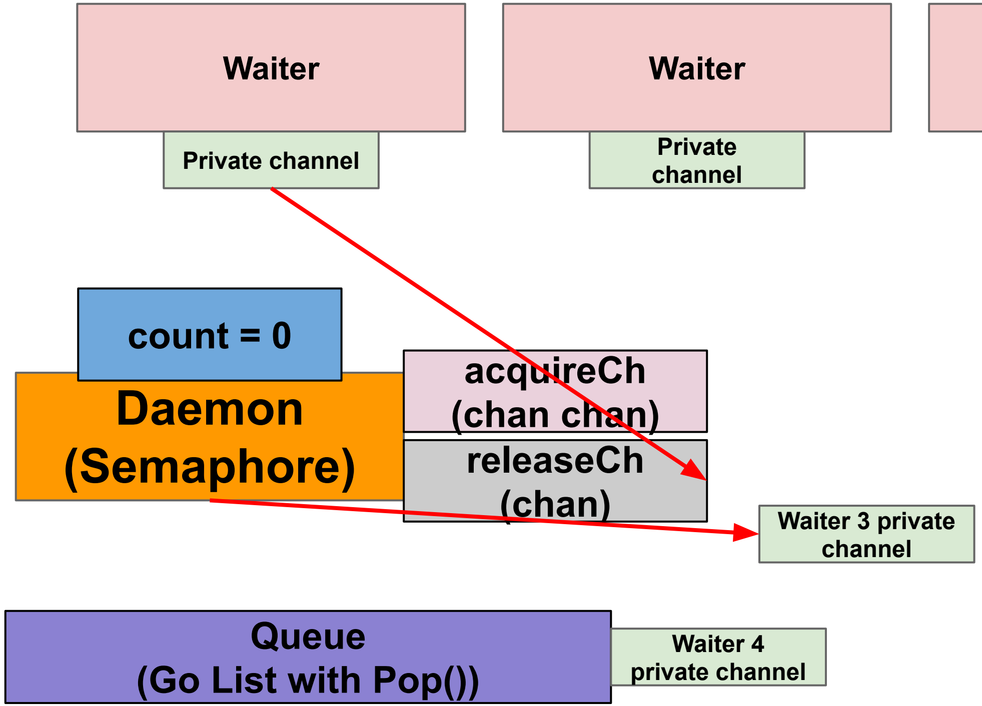



```go
case ch := <-sem.acquireCh: // Decrement or add a waiter
    if count > 0 {
        count--
        ch <- struct{}{} // Don't keep waiter blocked
    } else {
        waiters.PushBack(ch) // Add waiter to back of queue
    }
}
```

Go: FIFO Daemon Solution

```
func (s *Semaphore2) Acquire() {
    ch := make(chan struct{})
    // Send daemon a channel that can be used to unblock us
    s.acquireCh <- ch
    // Block until daemon decides to unblock us
    <-ch
}
```

- Anyone can signal
- And oldest waiter will be **written to** & **released**!




```
select {
case <-sem.releaseCh: // Increment or unblock a wai
    if waiters.Len() > 0 {
        ch := waiters.Pop()
        ch <- struct{}{} // Unblocks the oldest wai
    } else {
        count++
    }
```

Summary

- Invariants through Mutexes / condvars / etc vs channels-only [**shared vs distributed memory**]

- More use of the **chan chan** concept: goroutines ***pushing*** an indicator that they want to / are ready to do something

- **Note: start learning Rust!**