CS3211 Tutorial 7

Classic Concurrency Problems in C++ & Go

Simon

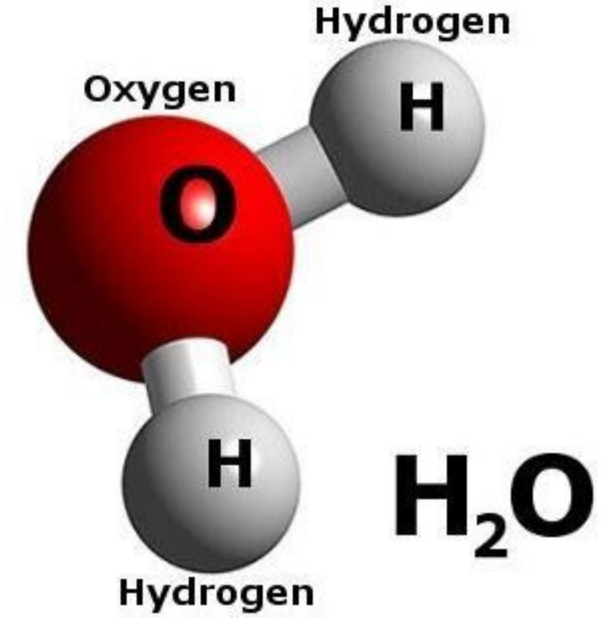
Adapted from Sriram's and Xue Yong's Slides

H2O Problem

i.e. resource management

1. H2O Problem

- Hydrogen and oxygen atoms arrive at the factory.
- **We only want to bond 2 H with 1 O.**
- Atoms only start bonding when 2 H and 1 O arrives.
- Other atoms block.
- Only 1 bonding can occur at a time.



Models distributed resources allocation

Sister’s Problems: Cigarette Smokers Problem, River Crossing Problem.

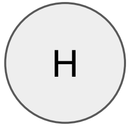
Source:
https://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf

H2O Problem: **Shared Memory**

i.e. resource management

1. H2O Problem

Incoming atoms | factory | completed



1. H2O Problem

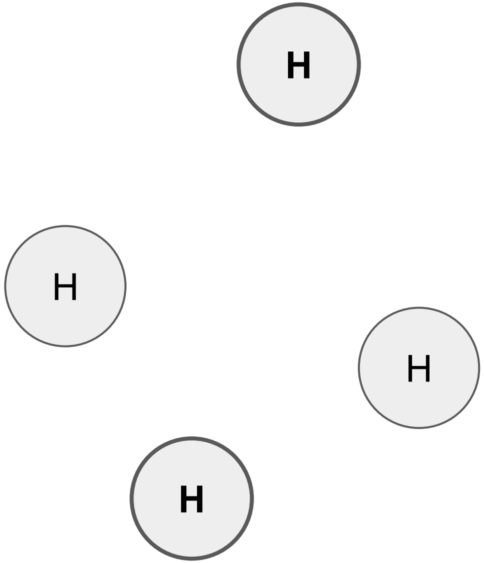
Incoming atoms

factory

completed

1. H2O Problem

Incoming atoms | factory | completed

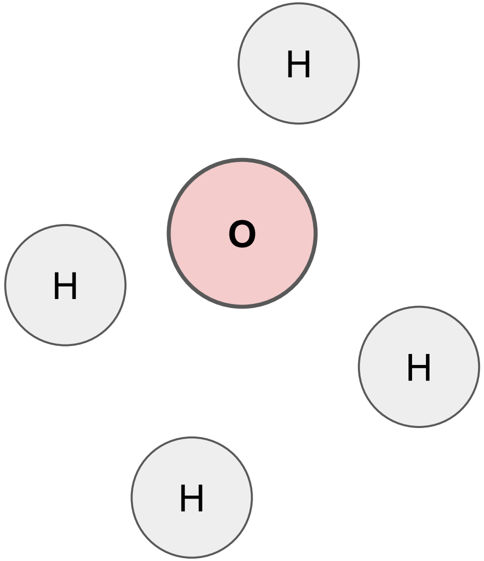


1. H2O Problem

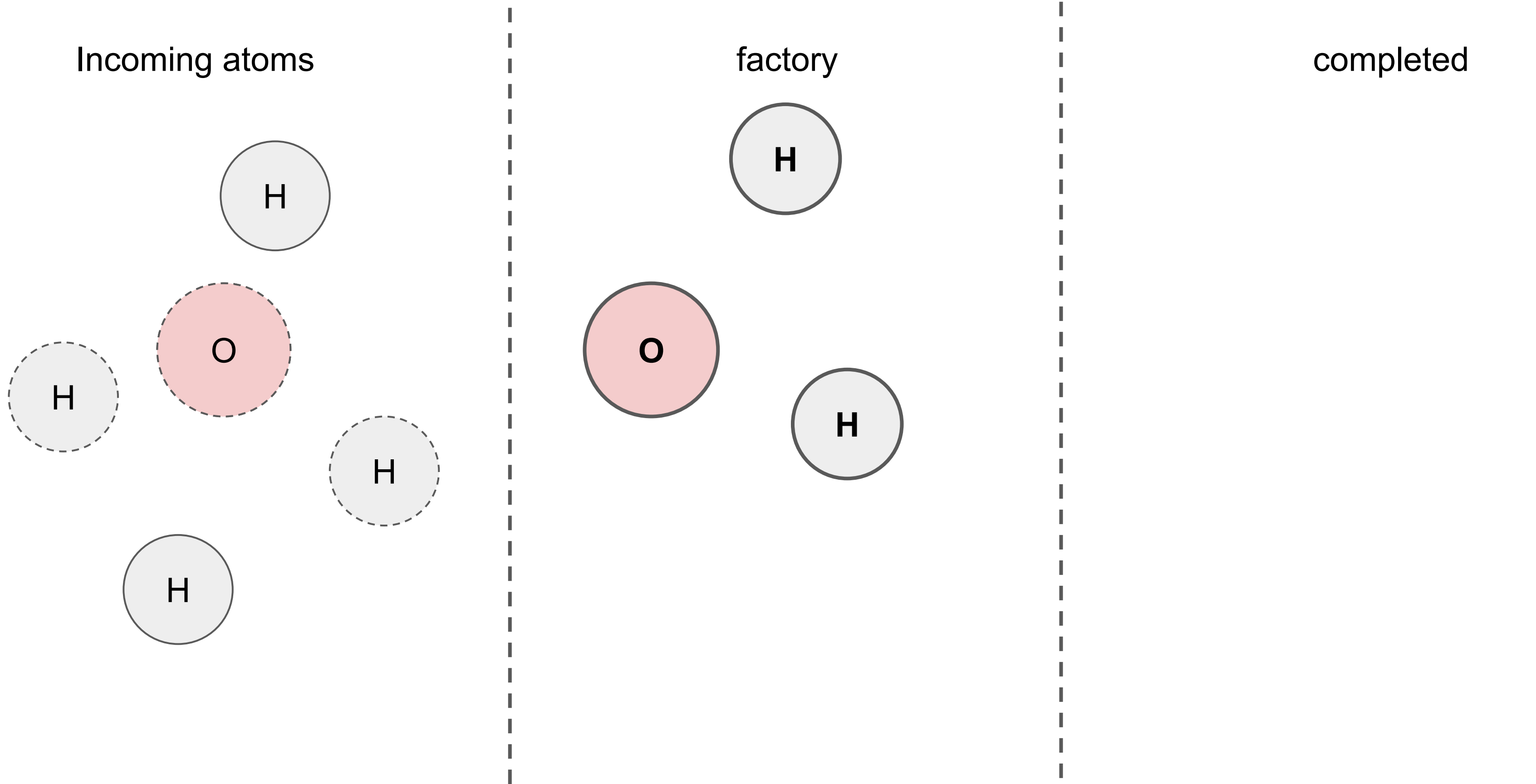
Incoming atoms

factory

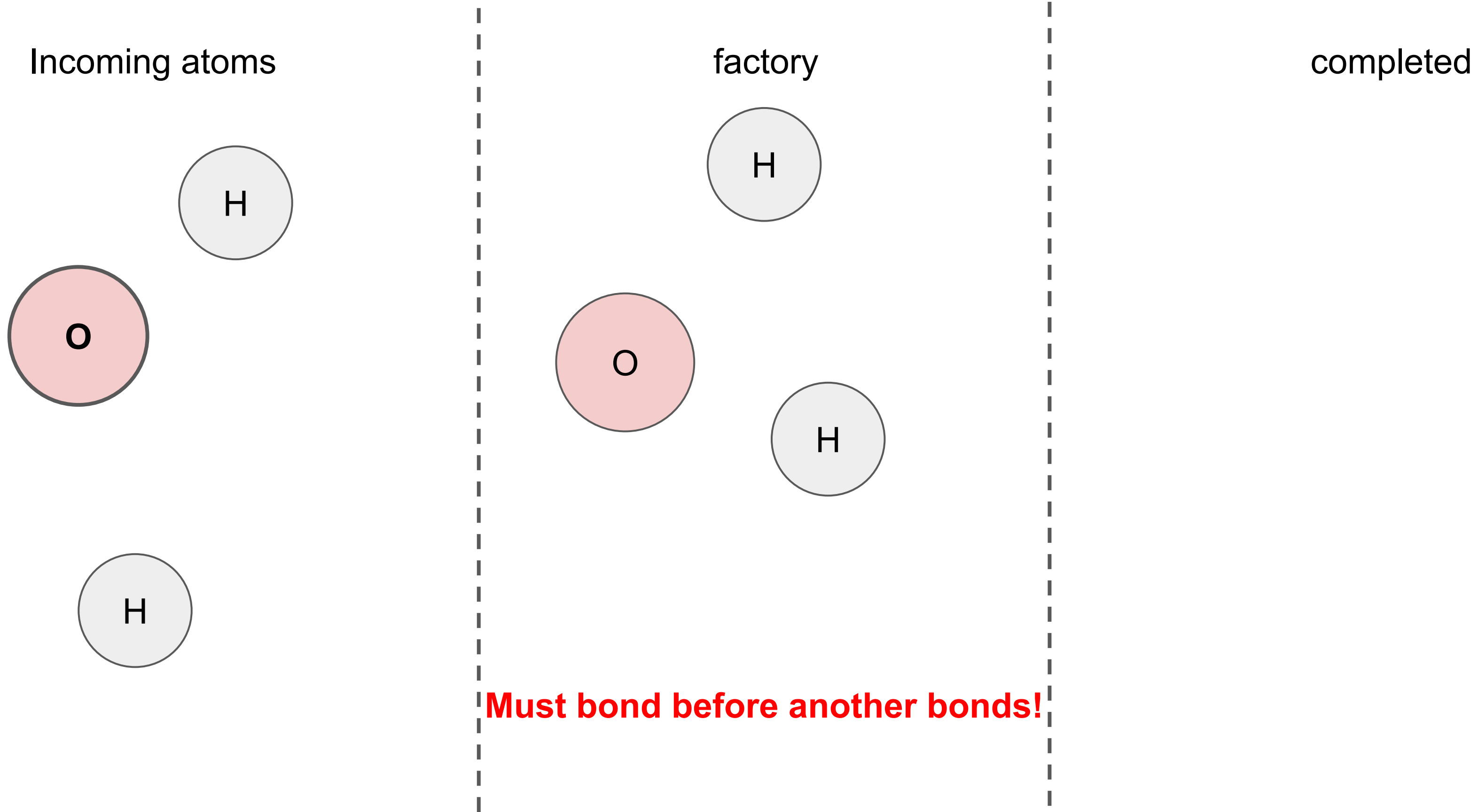
completed



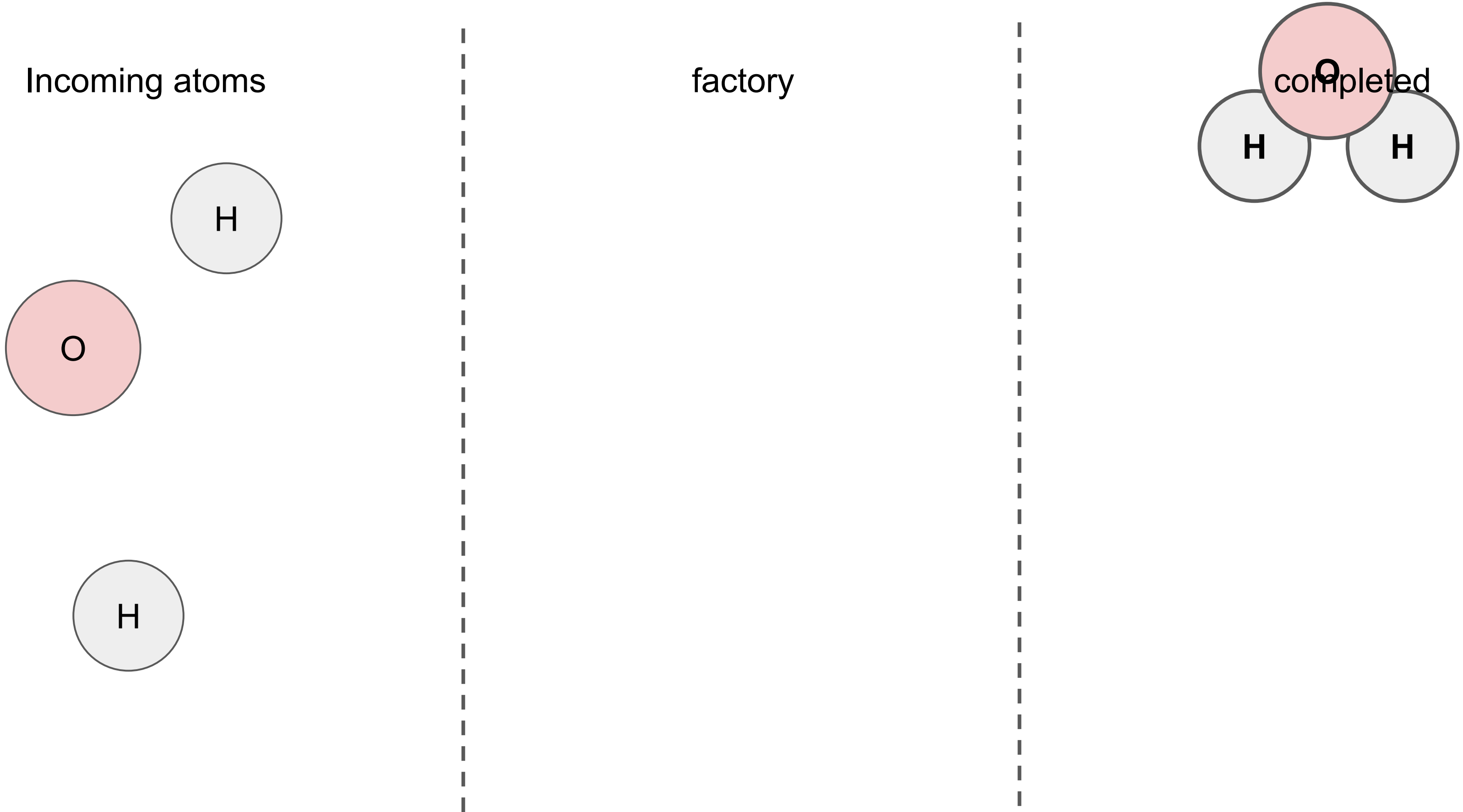
1. H2O Problem



1. H2O Problem



1. H2O Problem



Solution 1: Using a Barrier

Any problem here?

```
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();
  }
};
```

Solution 1: Using a Barrier

```
auto oxygen_bond = []() {
  print("Bonding oxygen\n");
  std::this_thread::sleep_for(std::chrono::milliseconds{5});
  print("Done\n");
};
std::thread t1{[&]() { factory.oxygen(oxygen_bond); }};
std::thread t2{[&]() { factory.oxygen(oxygen_bond); }};
std::thread t3{[&]() { factory.oxygen(oxygen_bond); }};
// Bonding oxygen
// Bonding oxygen
// Bonding oxygen
// Done
// Done
// Done
```

Does not ensure correct types! Here we form ozone rather than water

```
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();
  }
};
```

Solution 2: Using Semaphores (Extra)

Any problem here?

```
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();
  }
};
```

Solution 2: Using Semaphores (Extra)

Any problem here? **Yes!**

Case 1: 1 or 2 H atoms call bond() -> we get H2 or H

Case 2: 1 O atom call bond() -> We get O

```
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();
  }
};
```

Solution 3: Using Barrier + Semaphores

Any problem here?

```
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();
  }
};
```

Solution 3: Using Barrier + Semaphores

Any problem here? **NO!**

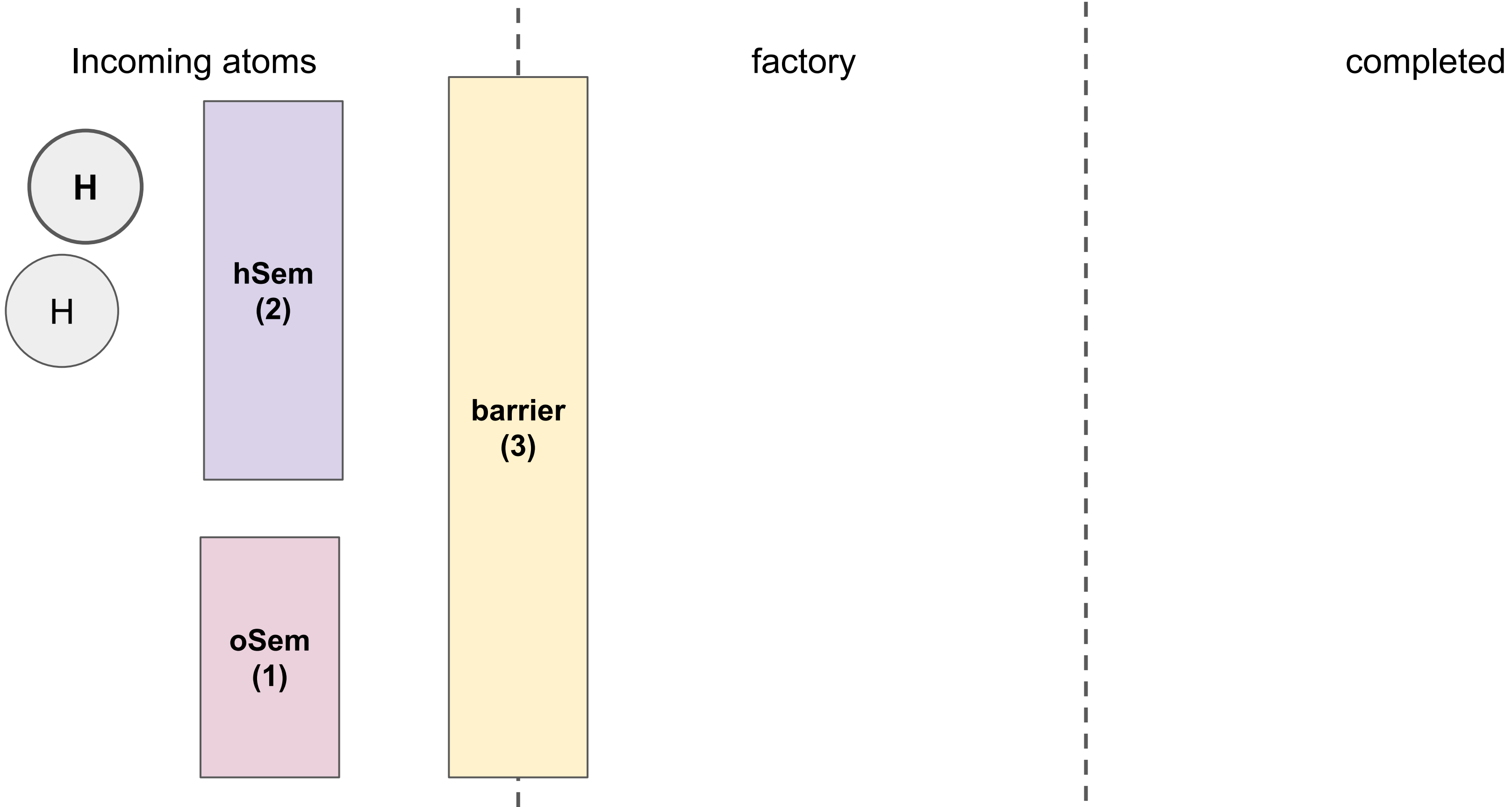
Case 1: there are less than 3 atoms -> atoms will block on barrier

Case 2: There are 3+ H atoms -> extra H will block on hSem

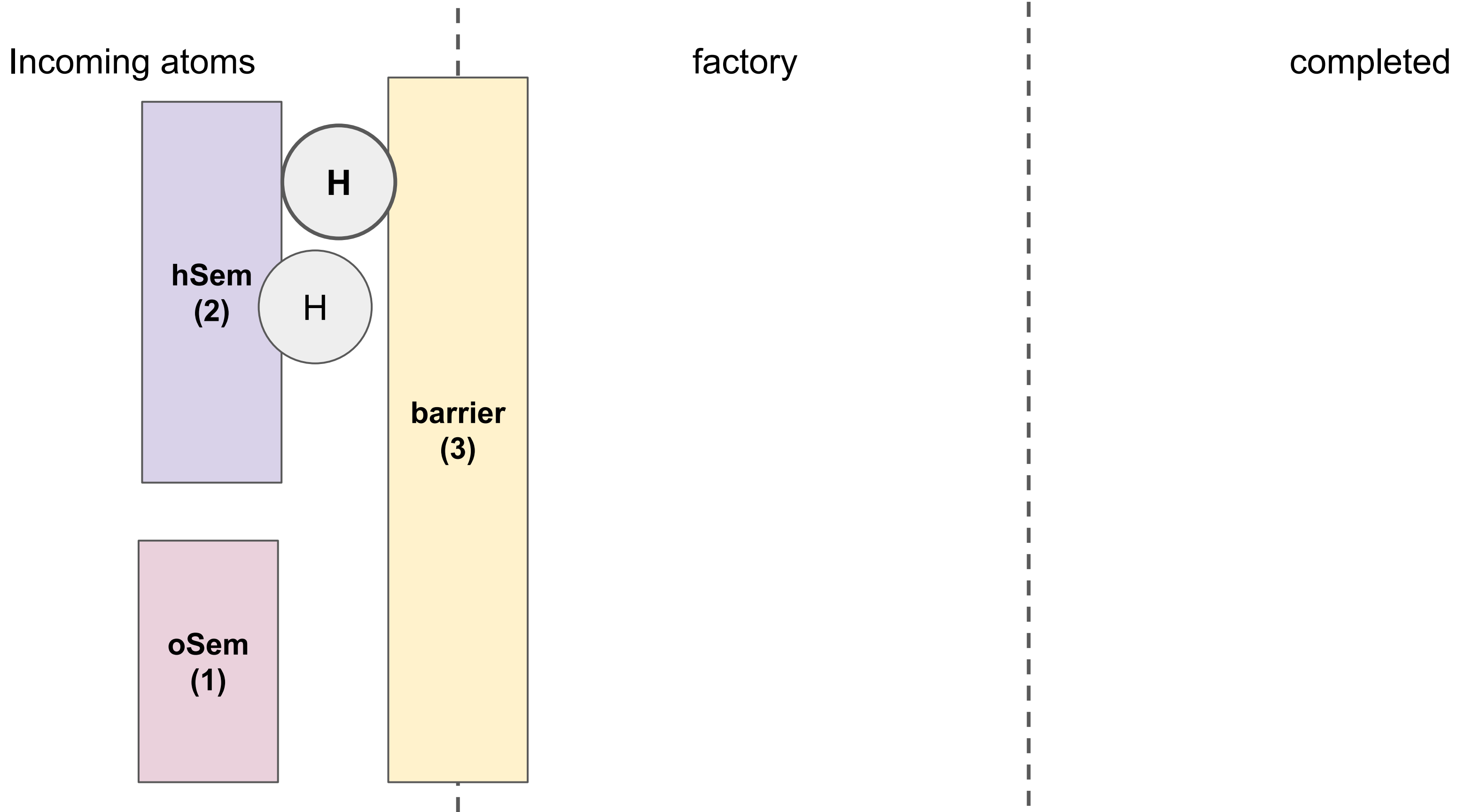
Case 3: There are 2+ O atoms -> extra O will block on oSem

```
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();
  }
};
```

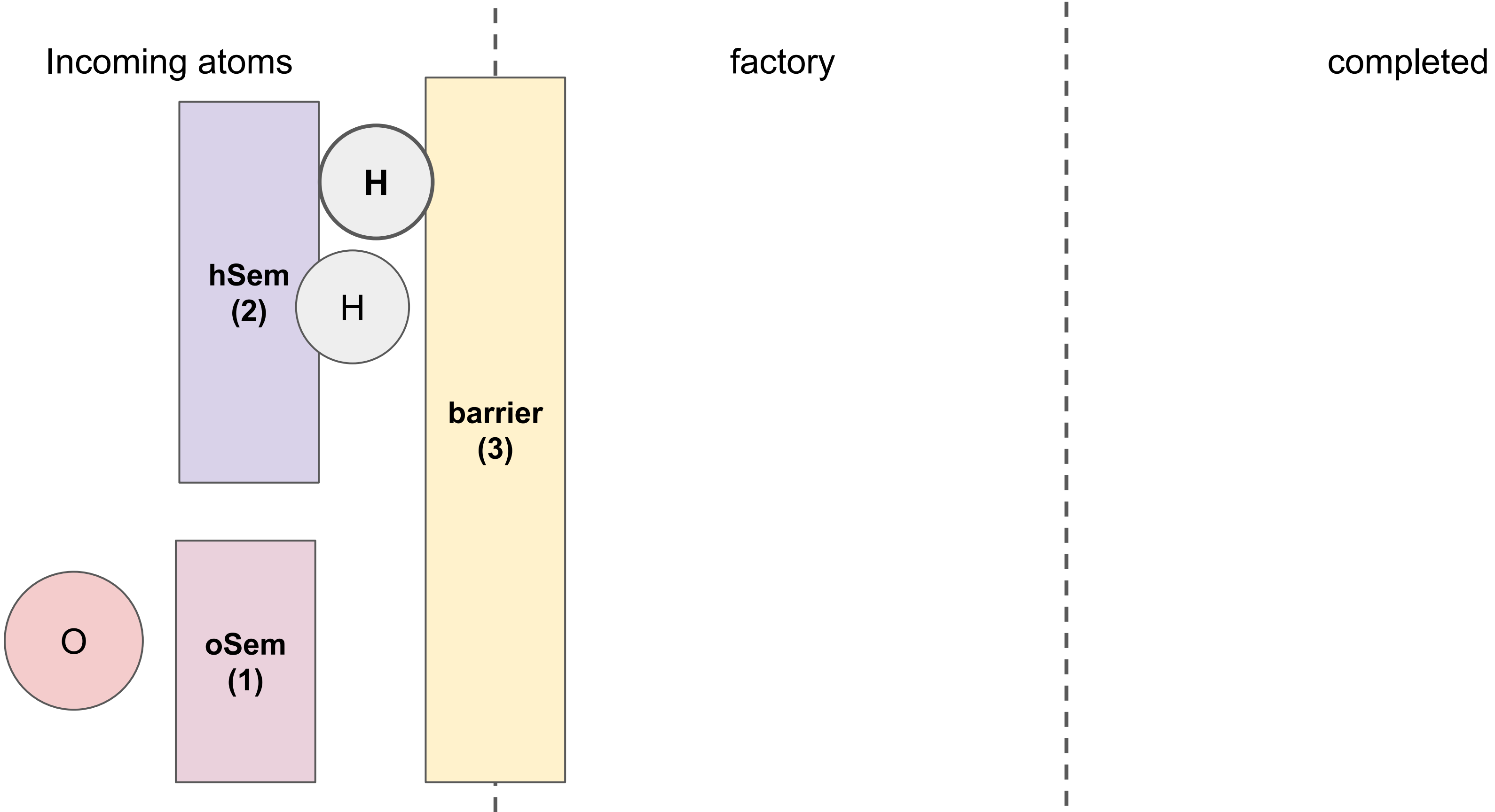
Solution 3: Using Barrier + Semaphores



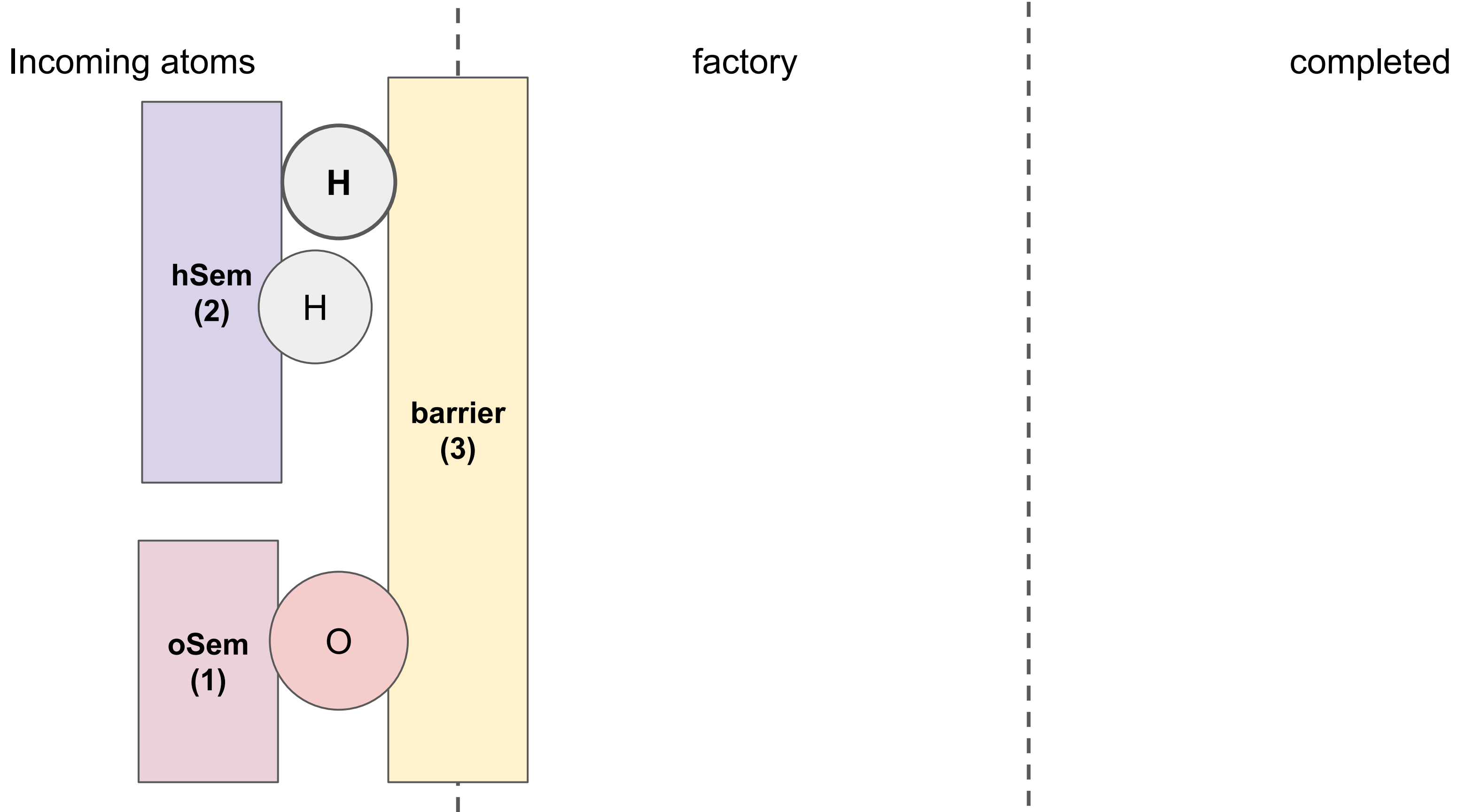
Solution 3: Using Barrier + Semaphores



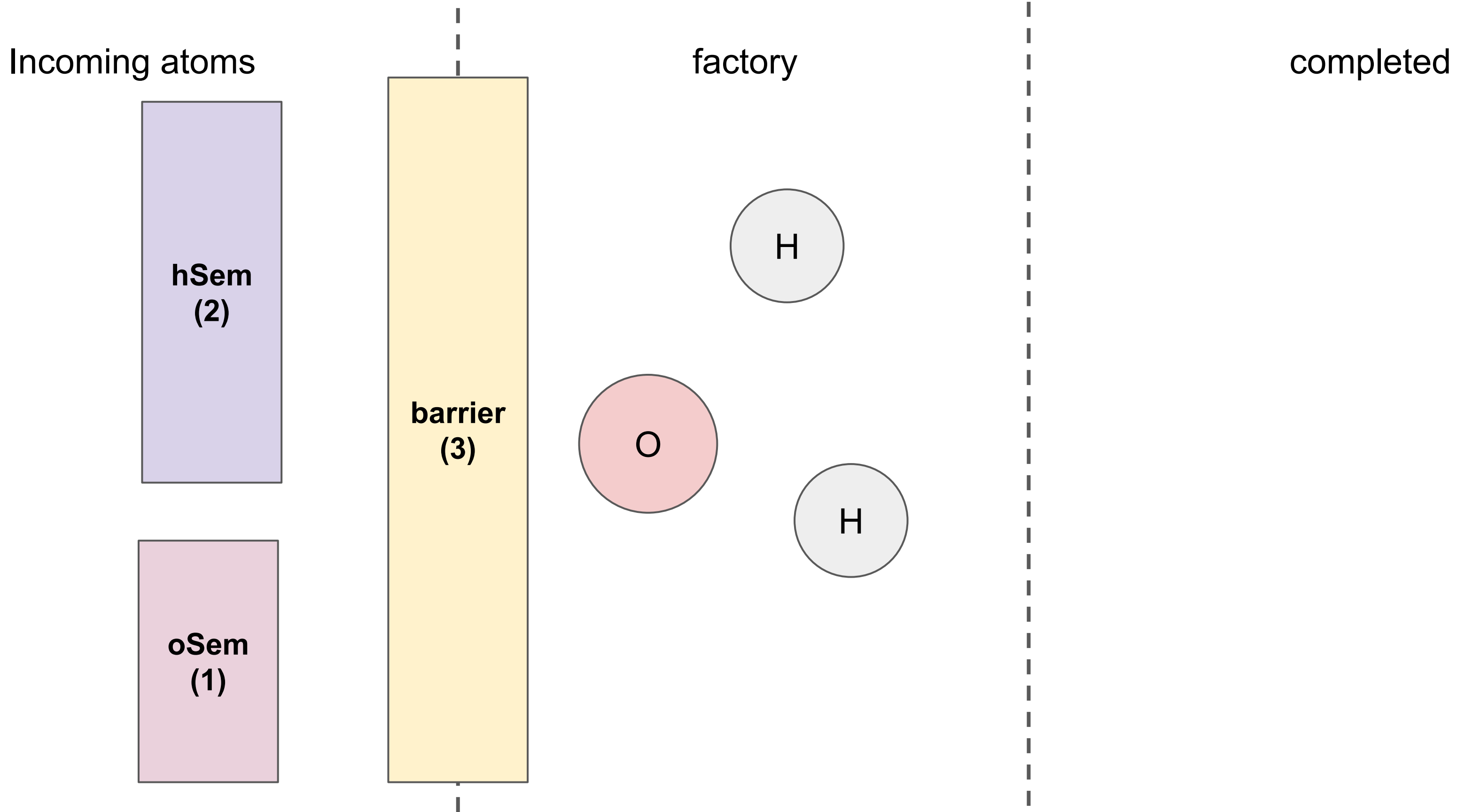
Solution 3: Using Barrier + Semaphores



Solution 3: Using Barrier + Semaphores



Solution 3: Using Barrier + Semaphores



Trivia

What if we use an `std::latch` instead of an `std::barrier`?

Trivia

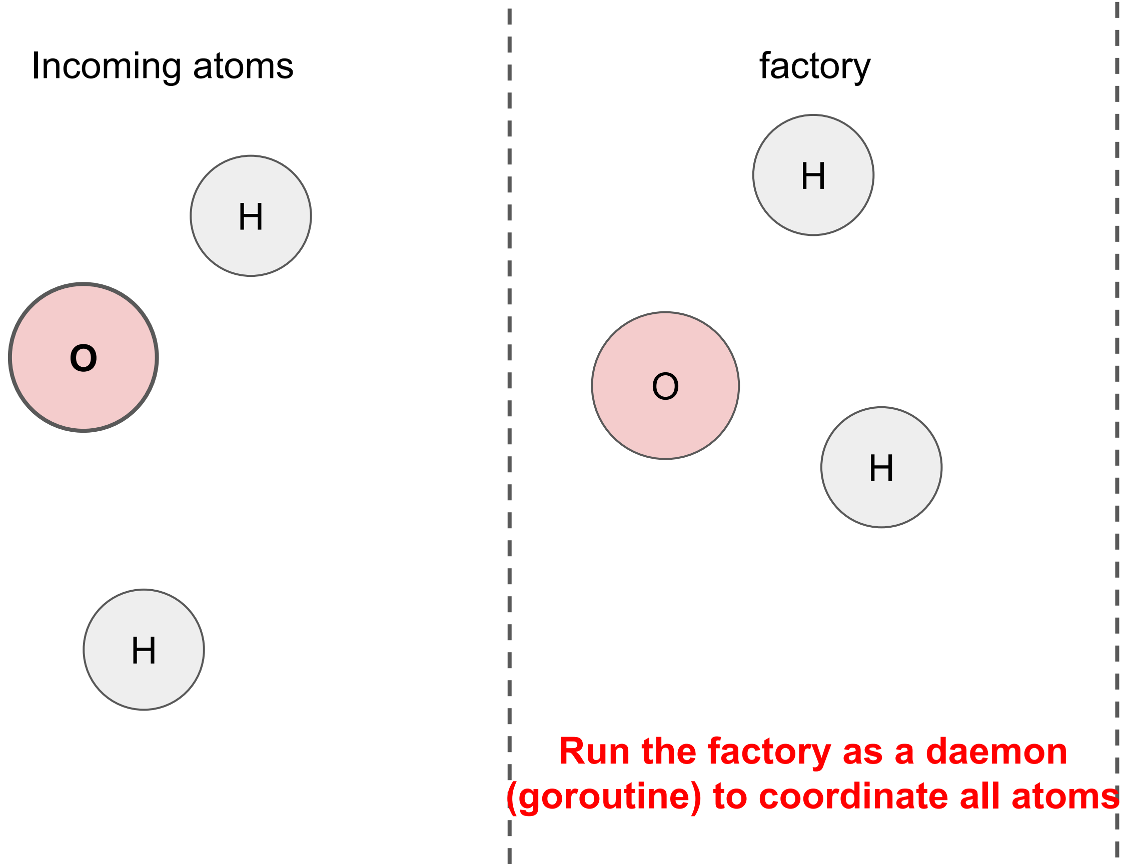
What if we use an `std::latch` instead of an `std::barrier`?

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.

H2O Problem: **Channels**

i.e. resource management

Go Solution 1:



Run the factory as a daemon (goroutine) to coordinate all atoms

Go Solution 1:

```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)
}
```

Go Solution 1:

Issues:

- daemon thread is a single point of failure

- potential performance bottleneck due to scheduling decisions

Go Solution 1:

Issues:

- daemon thread is a single point of failure
- potential performance bottleneck due to scheduling decisions

Recall: 1 oxygen per H2O, why not make oxygen the coordinator?

Go Solution 2:

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

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

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

    // 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{}{}
}
```

```go
type WaterFactoryWithLeader struct {
    oxygenMutex chan struct{}
    precomH     chan chan struct{}
}
```

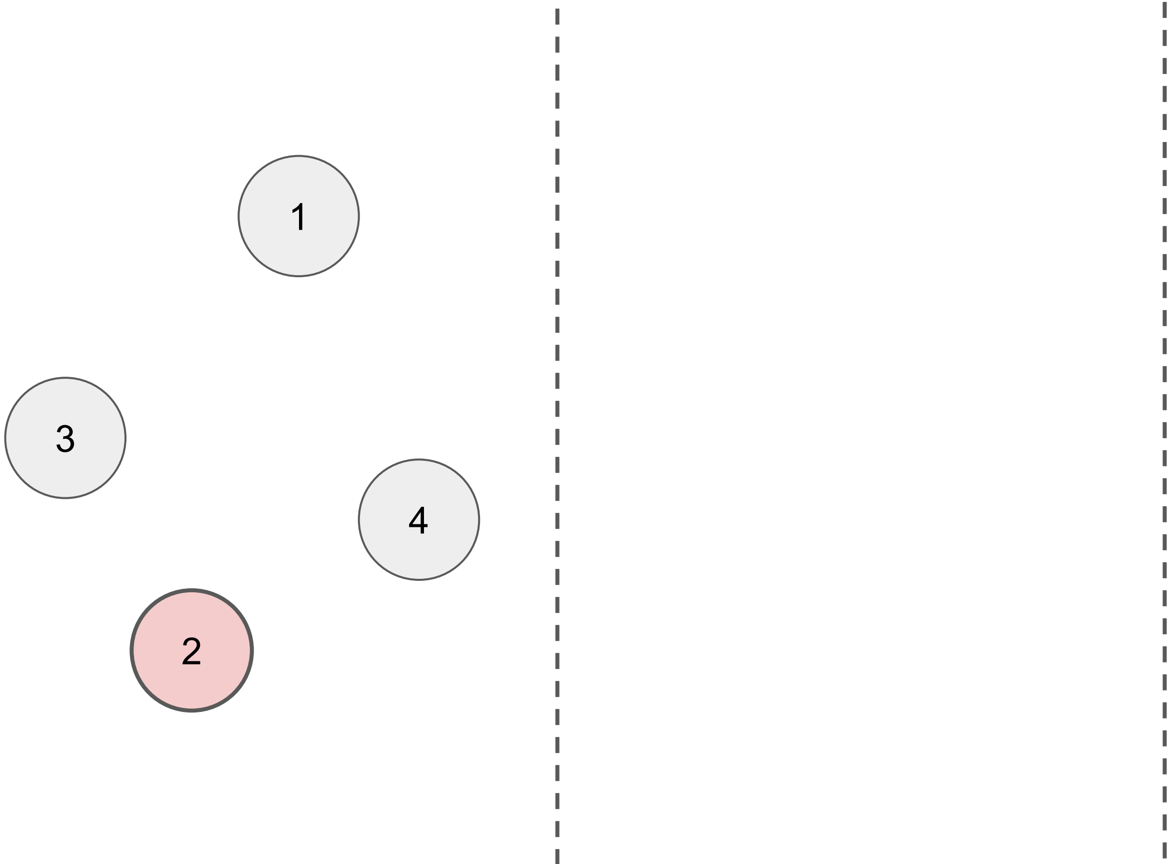
```go
func (wf *WaterFactoryWithLeader) hydrogen(bond func()) {
    commit := make(chan struct{}) // Step 1: Create private communication channel
    wf.precomH <- commit          // Step 2: (Precommit)
    <-commit                      // Step 3: (Commit)
    bond()                        // Step 4: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

Can still have issues if oxygen holding the “mutex” dies but the probability is lower

FIFO Semaphore

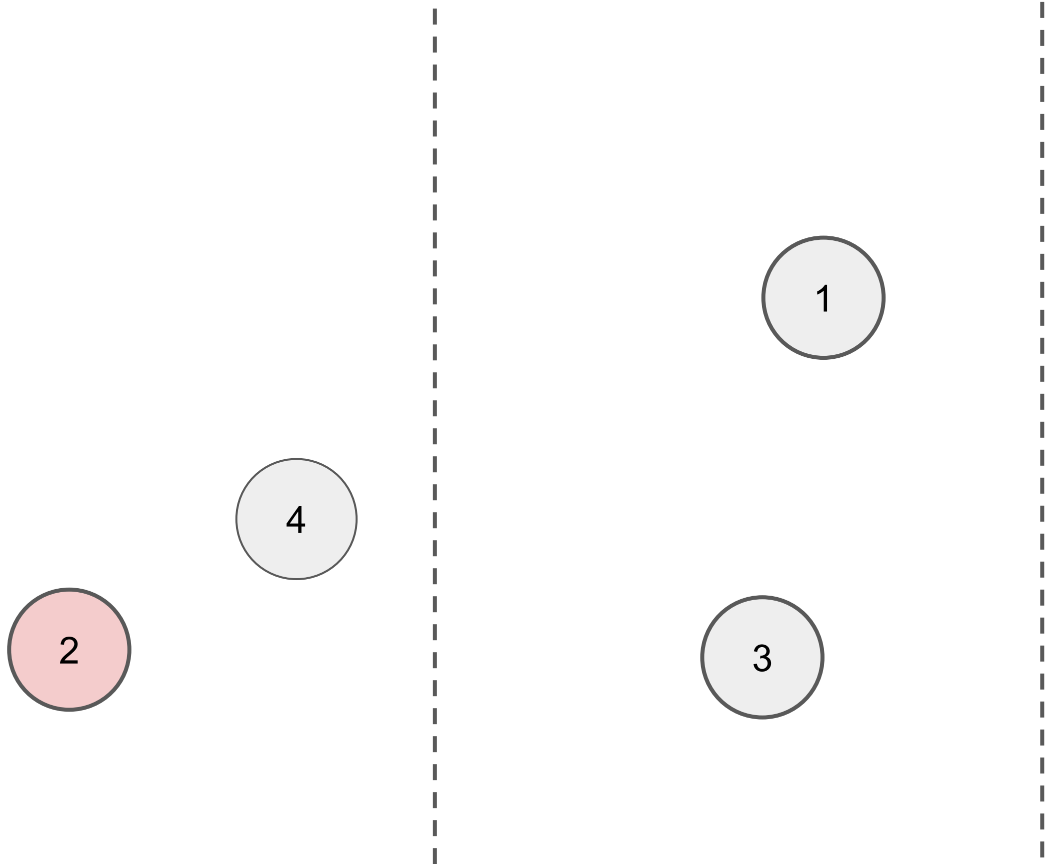
i.e. starvation-free

FIFO Semaphore(=2) Problem Statement



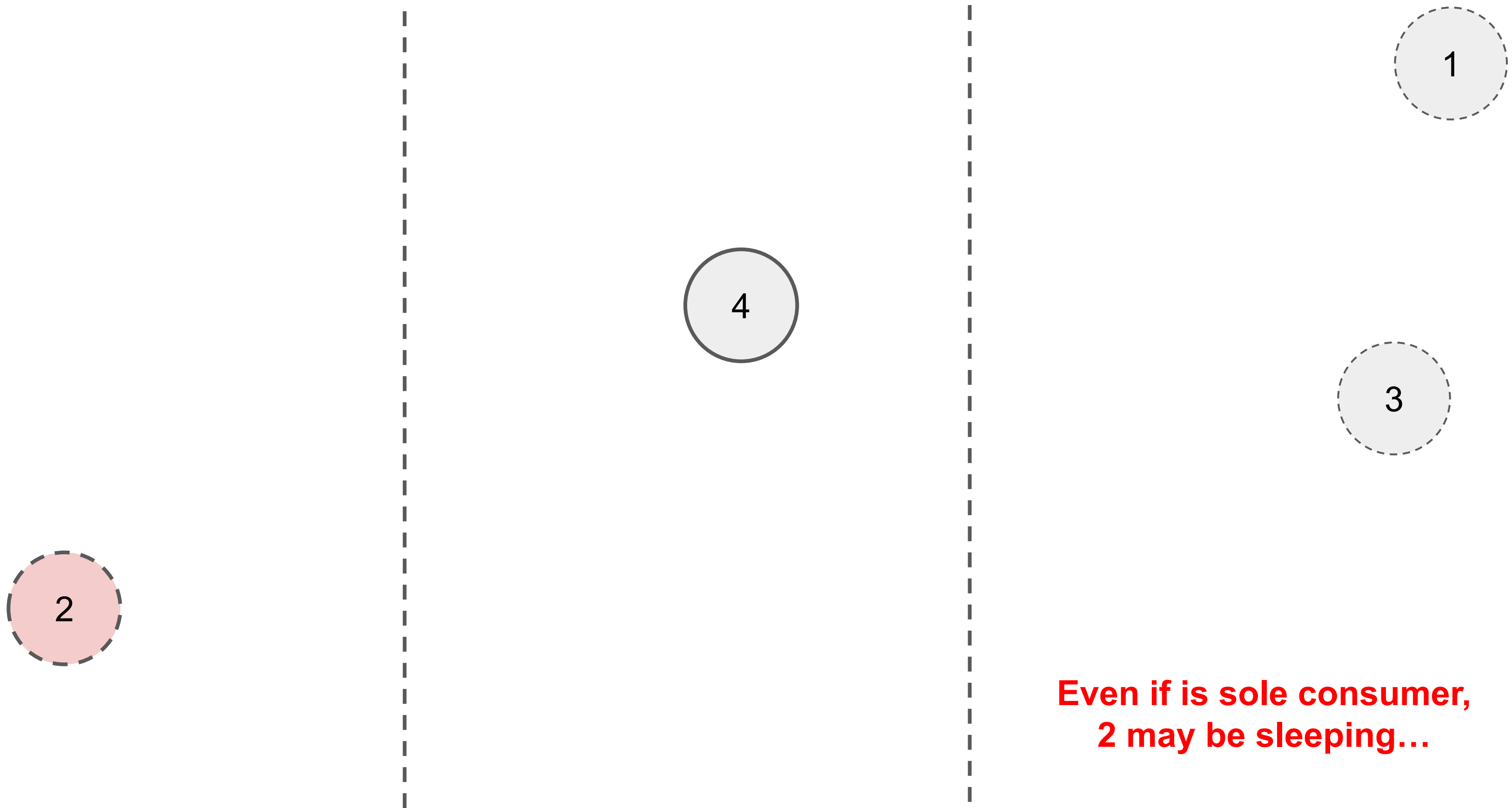
E.g. Oxygen atoms waiting to bond()

FIFO Semaphore(=2) Problem Statement



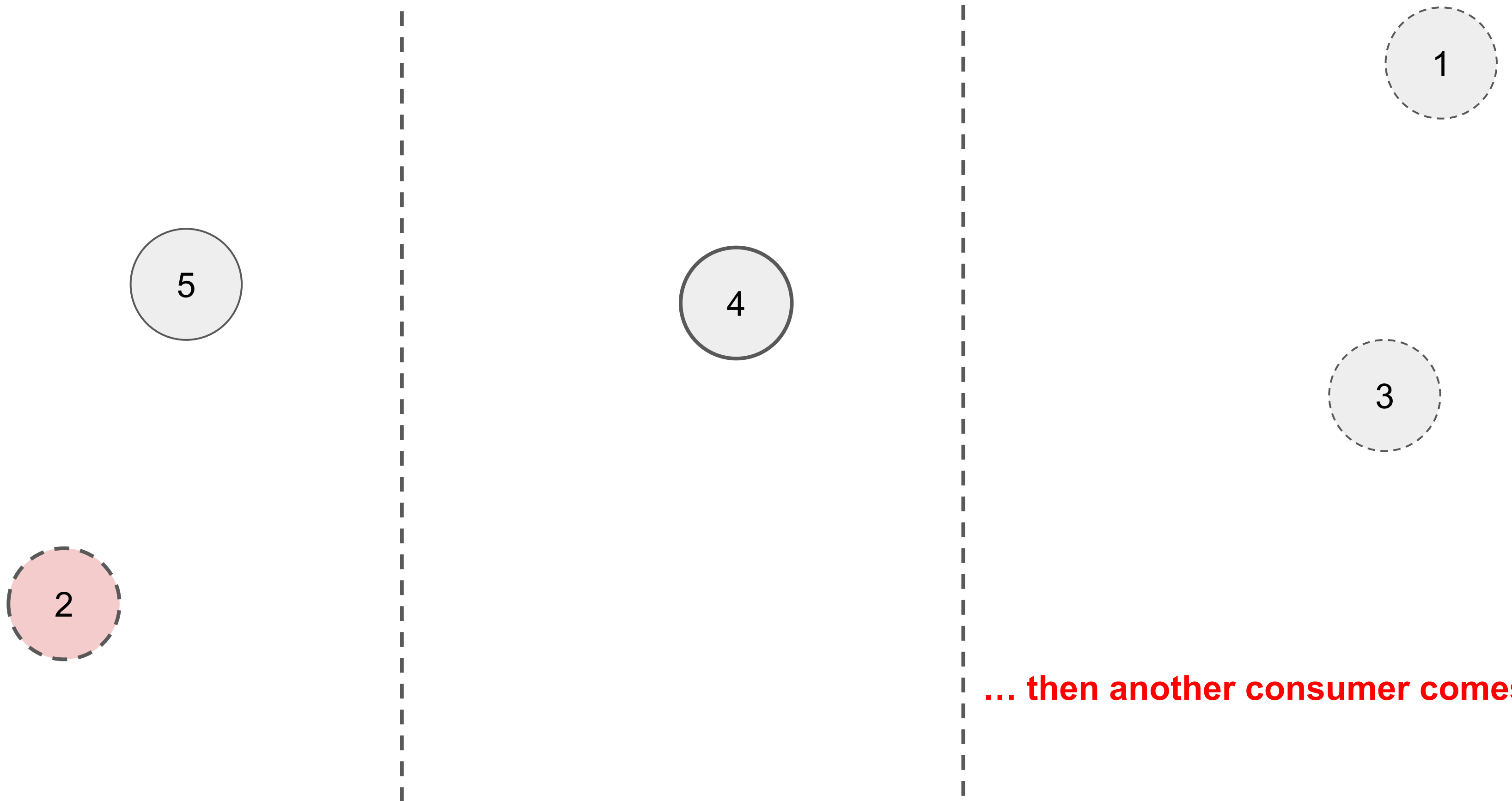
Semaphores don't guarantee order!

FIFO Semaphore(=2) Problem Statement



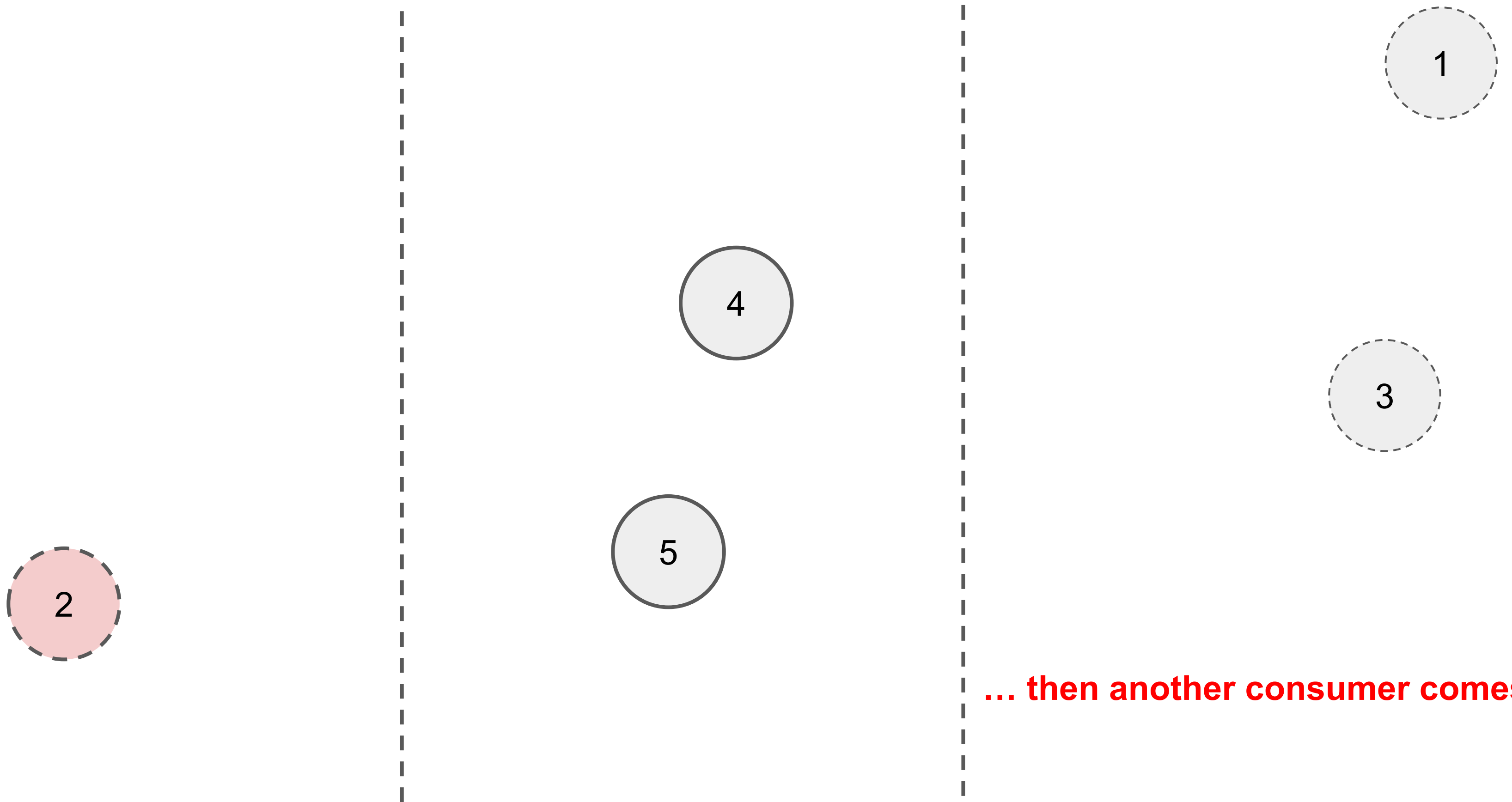
Even if is sole consumer,
2 may be sleeping…

FIFO Semaphore(=2) Problem Statement



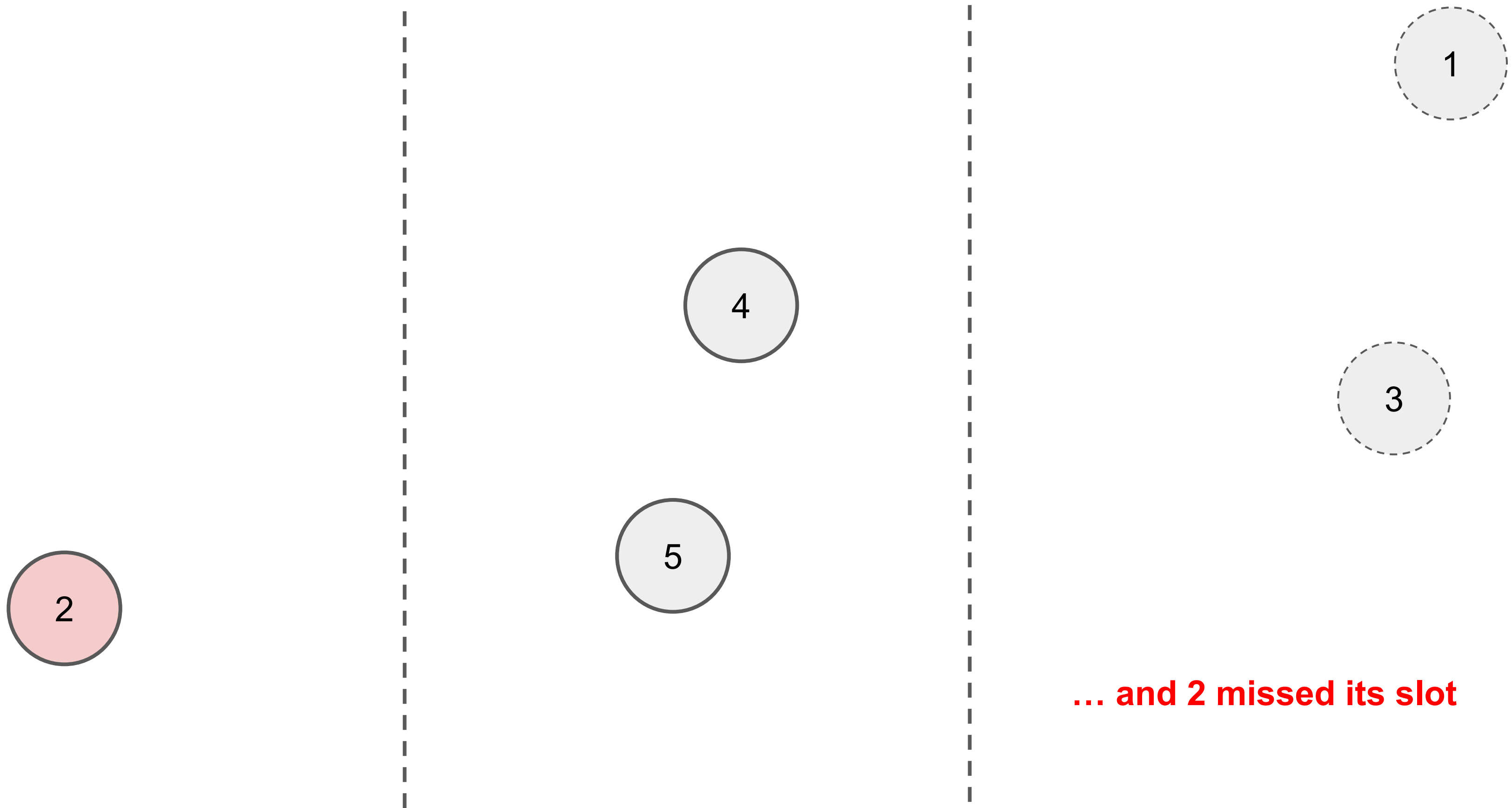
… then another consumer comes

FIFO Semaphore(=2) Problem Statement



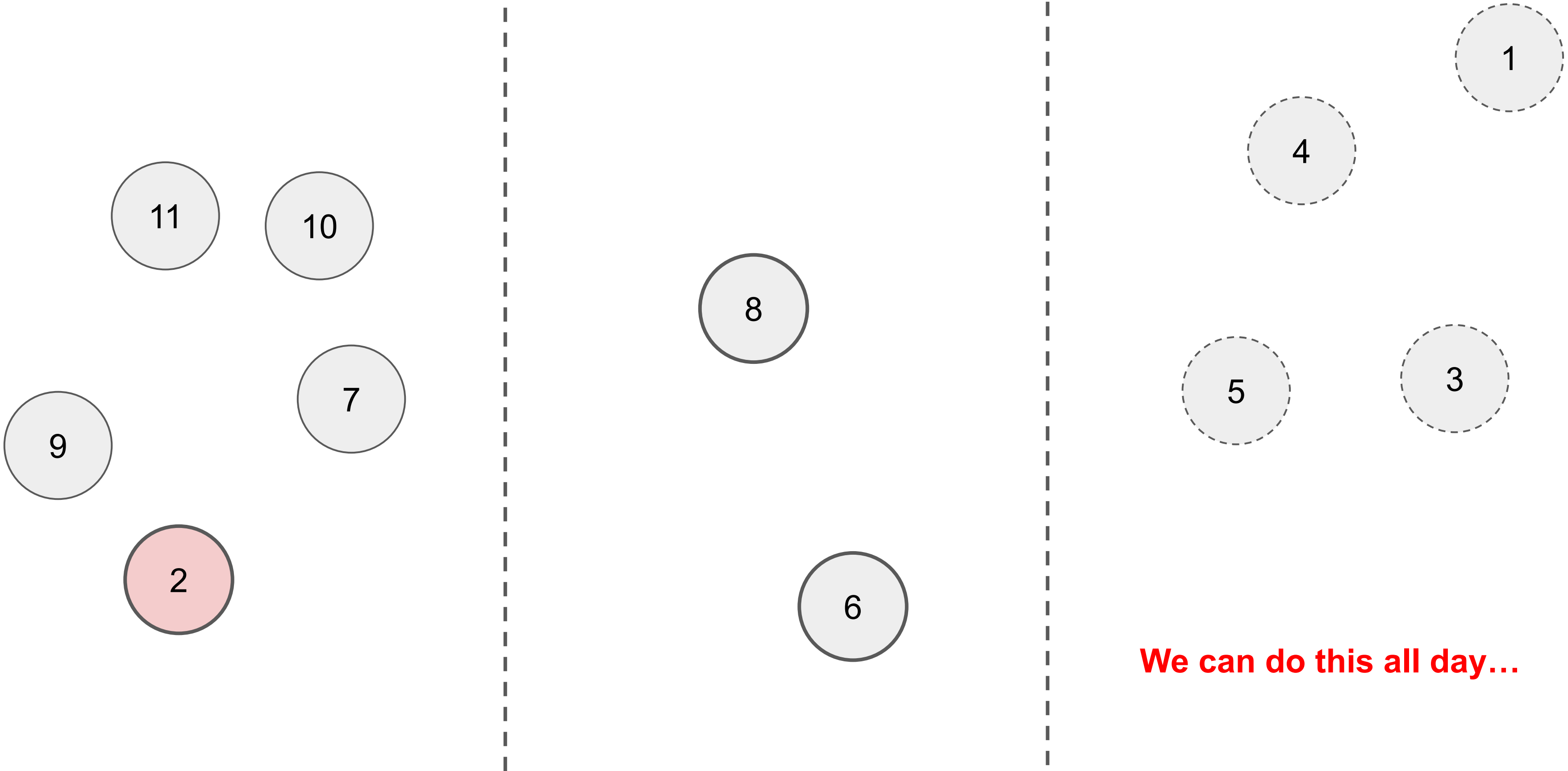
… then another consumer comes

FIFO Semaphore(=2) Problem Statement



… and 2 missed its slot

FIFO Semaphore(=2) Problem Statement



We can do this all day…

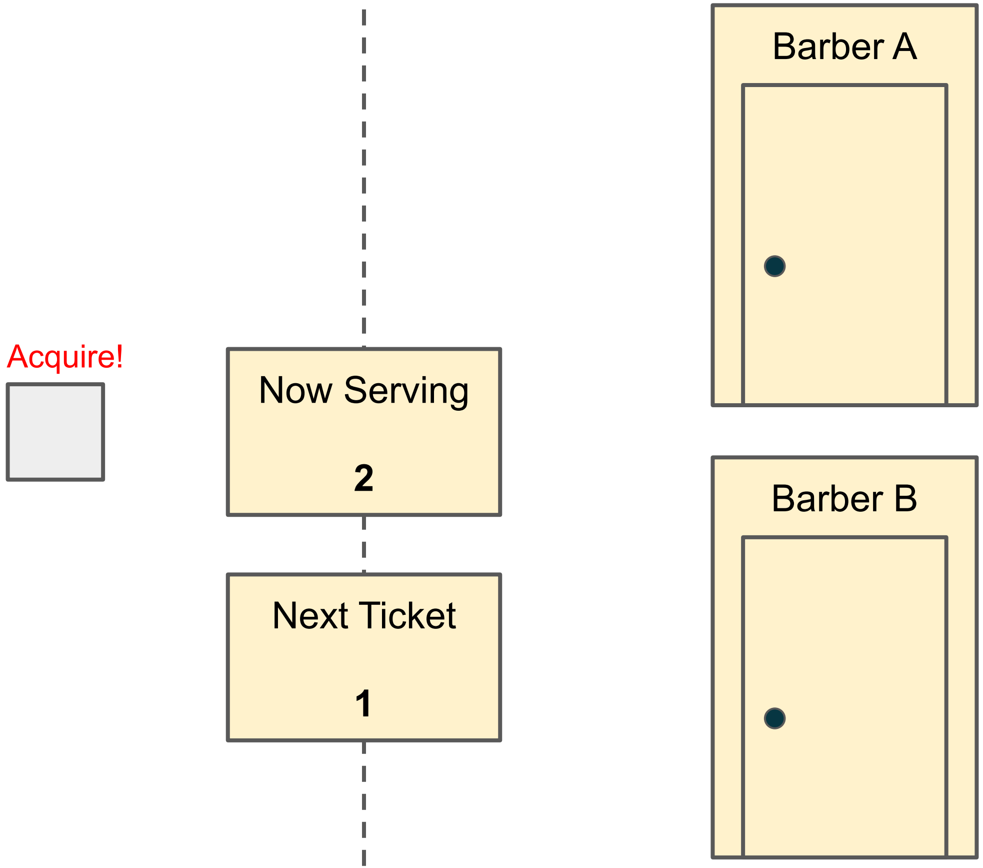
FIFO using ticket queue

Demo 4

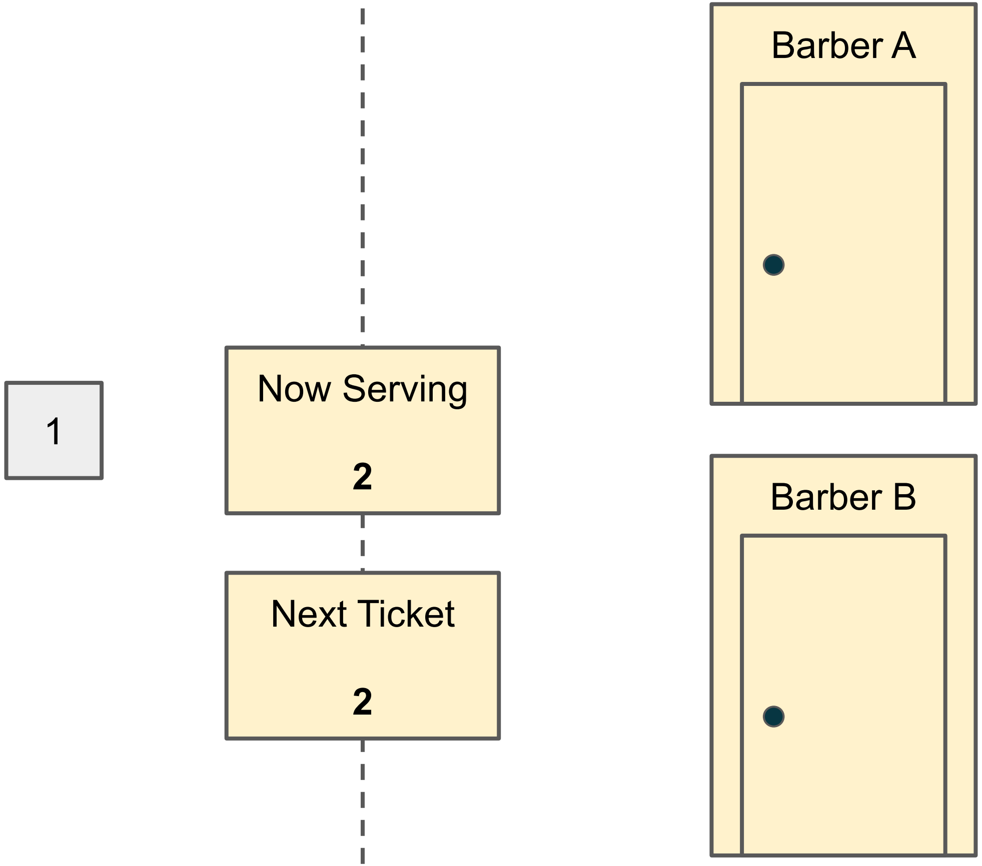
Demo 4: ticket queue

```
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);
  }
};
```

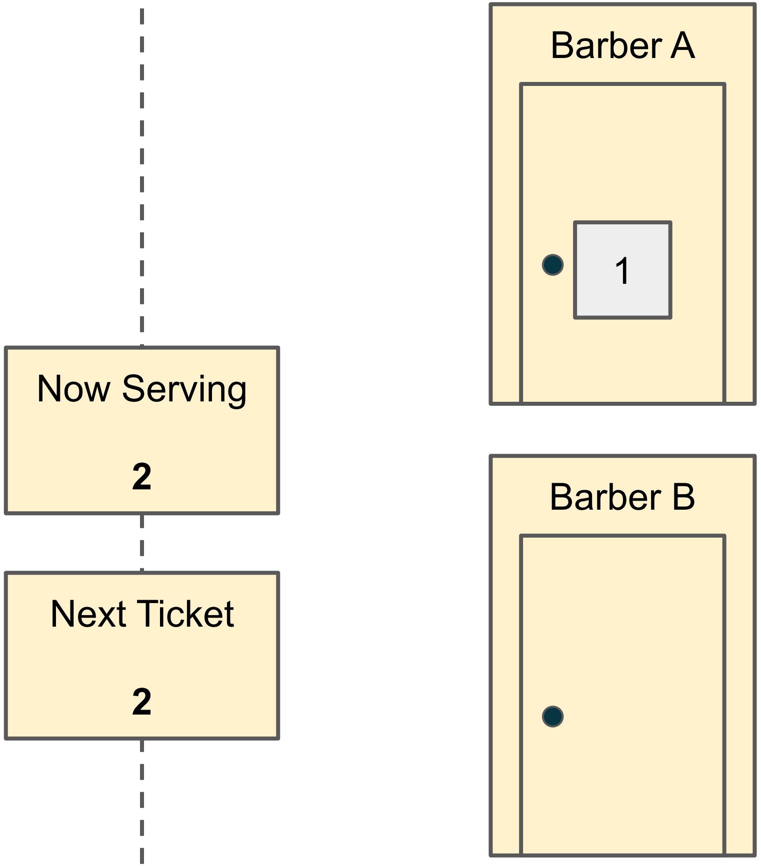
Demo 4: ticket queue (initial_count = 2)



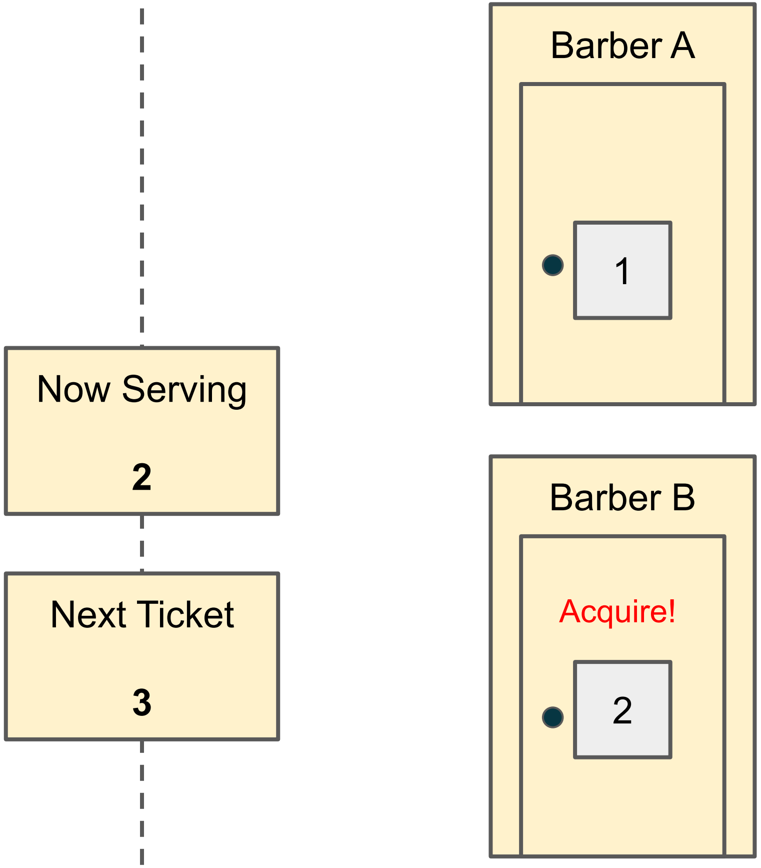
Demo 4: ticket queue (initial_count = 2)



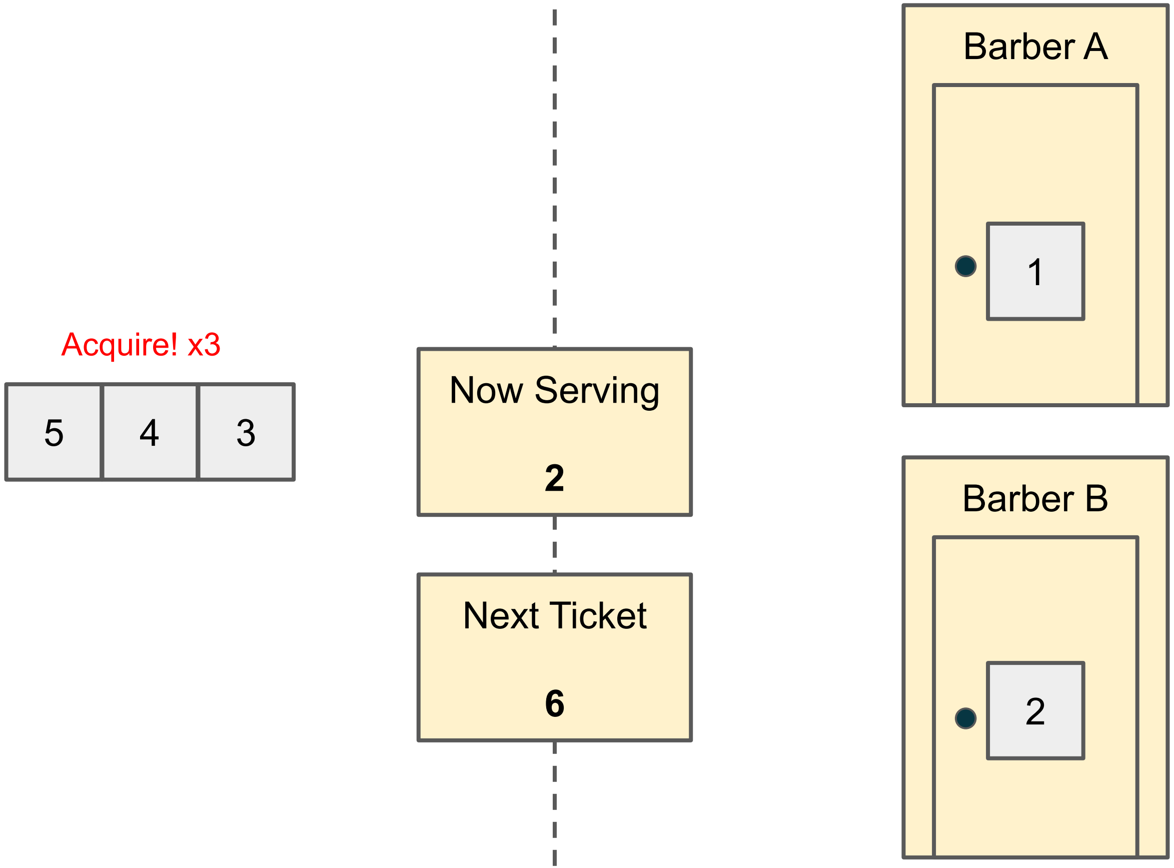
Demo 4: ticket queue (initial_count = 2)



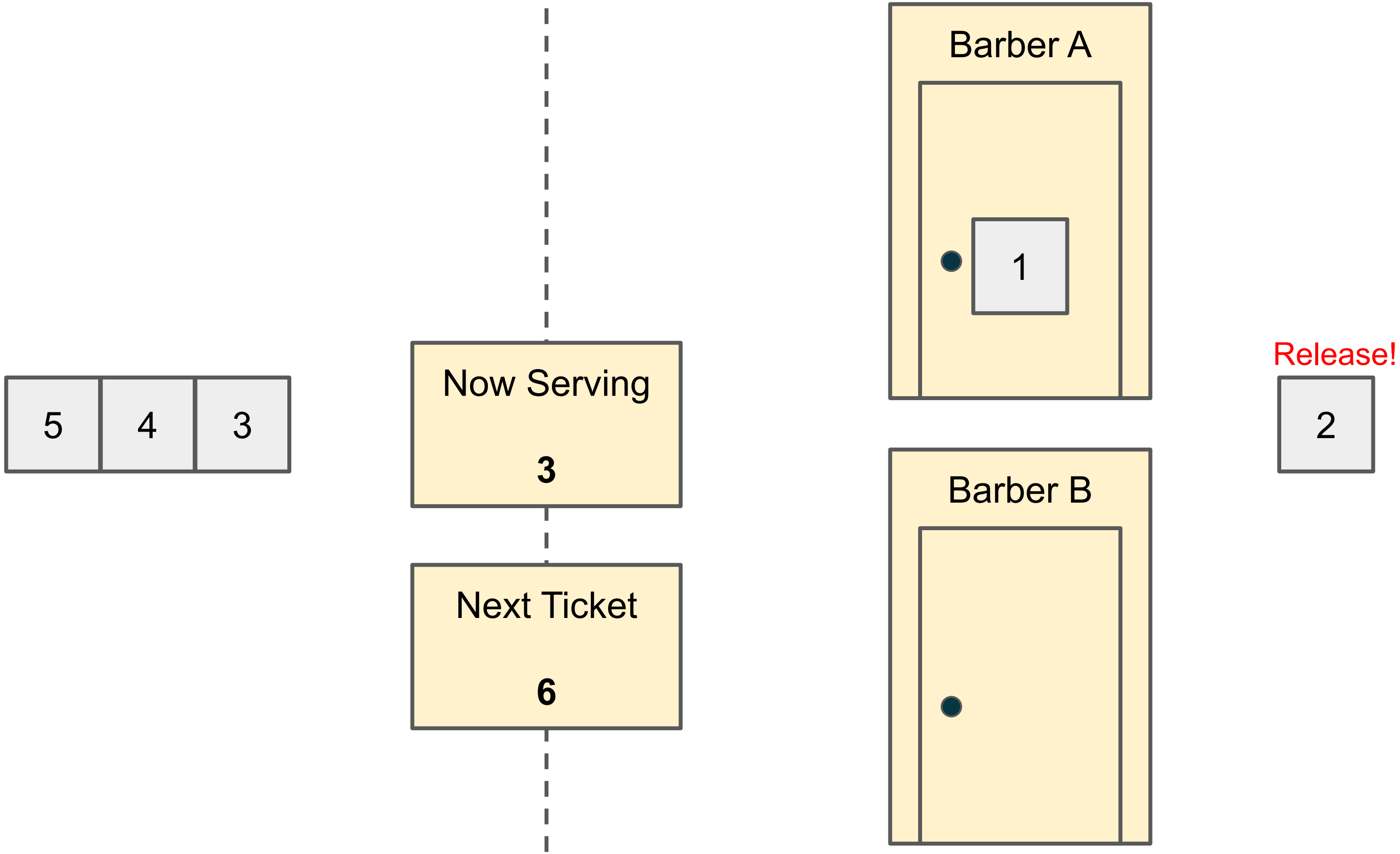
Demo 4: ticket queue (initial_count = 2)



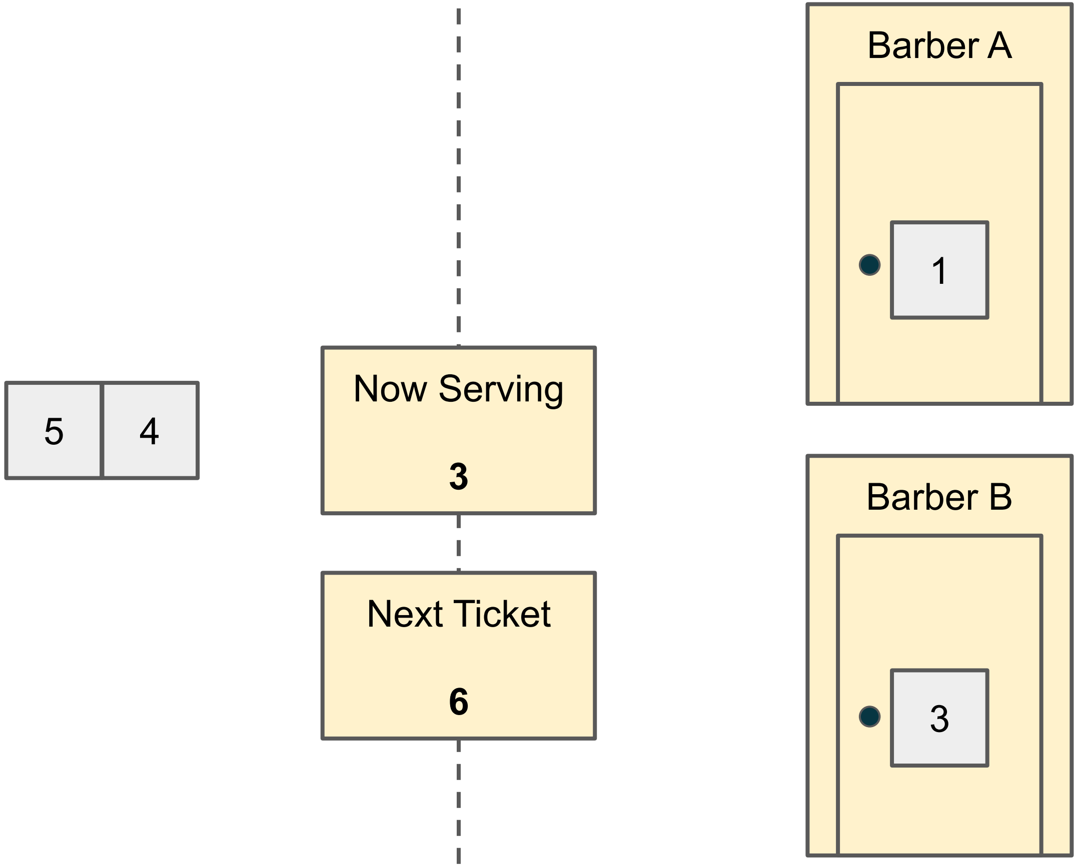
Demo 4: ticket queue (initial_count = 2)



Demo 4: ticket queue (initial_count = 2)



Demo 4: ticket queue (initial_count = 2)



Task 1: Replace spinwait

Try it yourself!

Use

condition_variable

or

atomic<T>::wait

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

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

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

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

Task 1: Replace spinwait

```
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);
    auto old_now_serving = now_serving.load();
    if(old_now_serving < my_ticket)
        now_serving.wait(old_now_serving);
  }

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

Any problems?

Task 1: Replace spinwait

Performs atomic waiting operations. Behaves as if it repeatedly performs the following steps:

- Compare the value representation of `this->load(order)` with that of `old`.
  - If those are equal, then blocks until `*this` is notified by `notify_one()` or `notify_all()`, or the thread is unblocked spuriously.
  - Otherwise, returns.

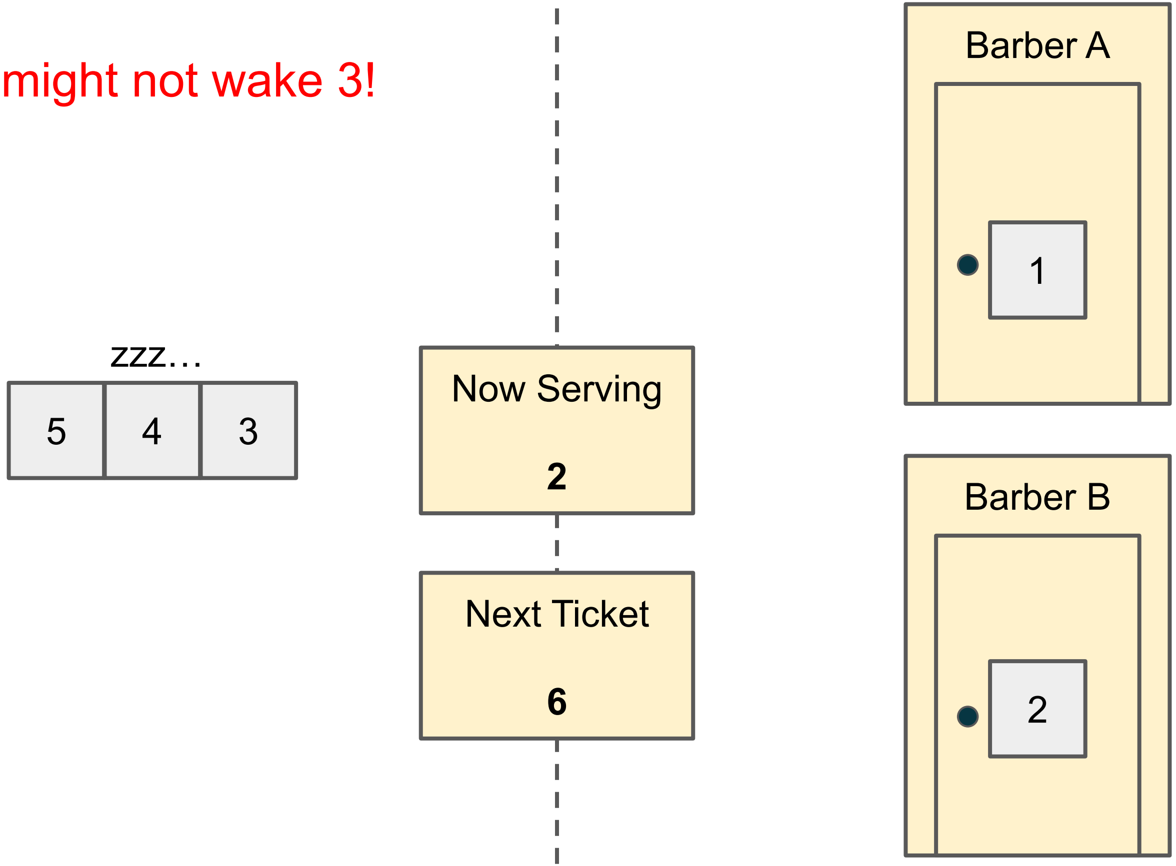
These functions are guaranteed to return only if value has changed, even if underlying implementation unblocks spuriously.

```
    if(old_now_serving < my_ticket)
      now_serving.wait(old_now_serving, my_ticket);
  }

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

Just because old changed doesn't mean the condition is true

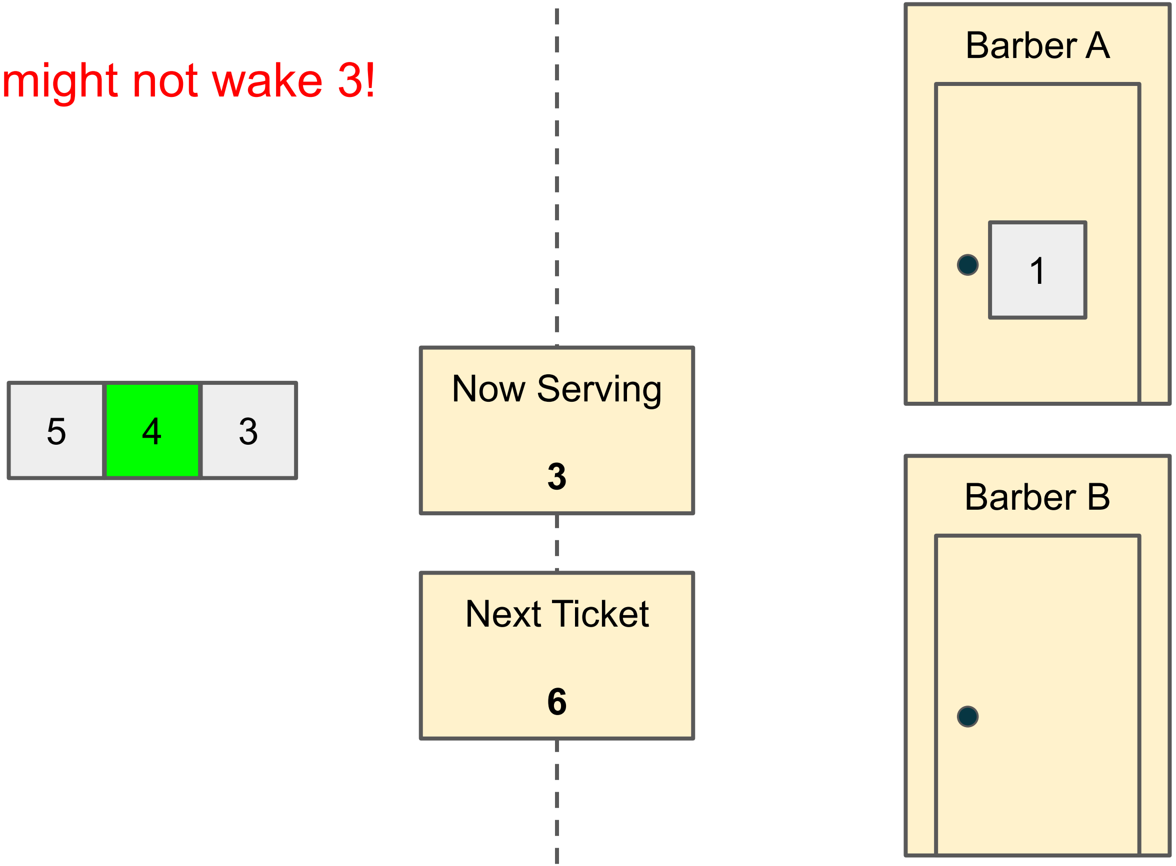
Task 1: Replace spinwait

`notify_one()` might not wake 3!



Task 1: Replace spinwait

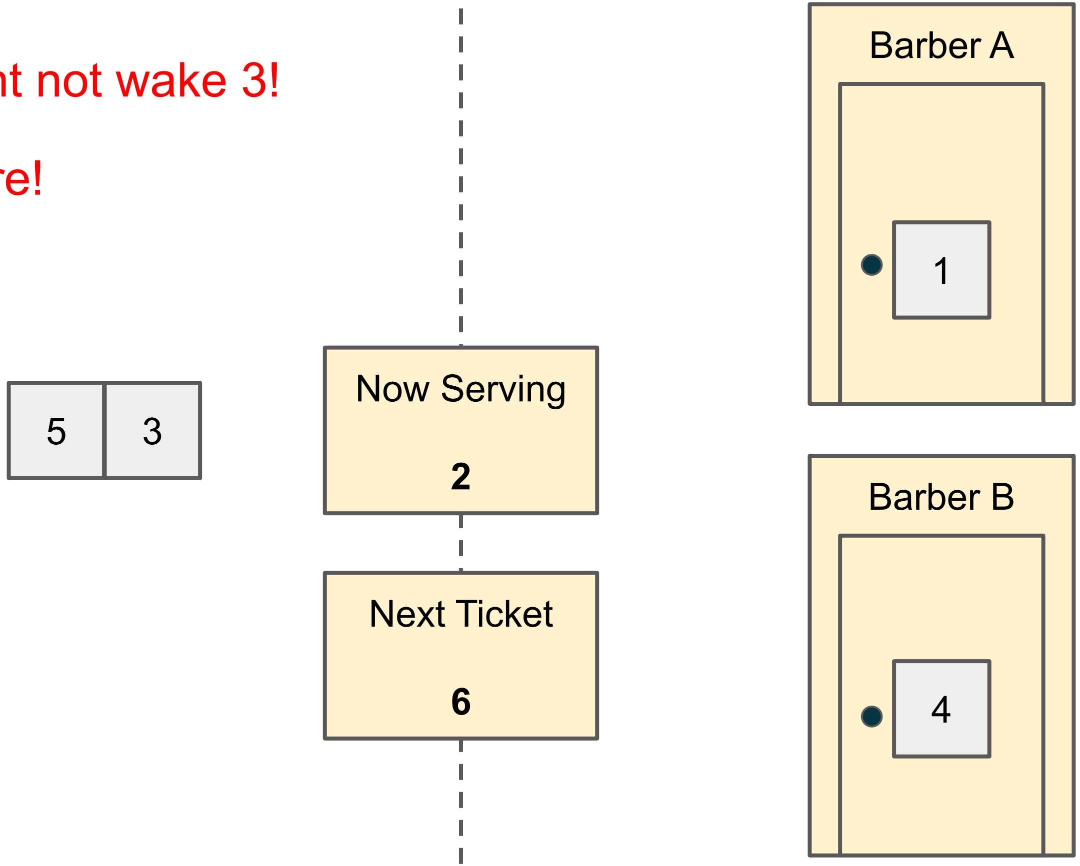
`notify_one()` might not wake 3!



Task 1: Replace spinwait

`notify_one()` might not wake 3!

-> not FIFO anymore!



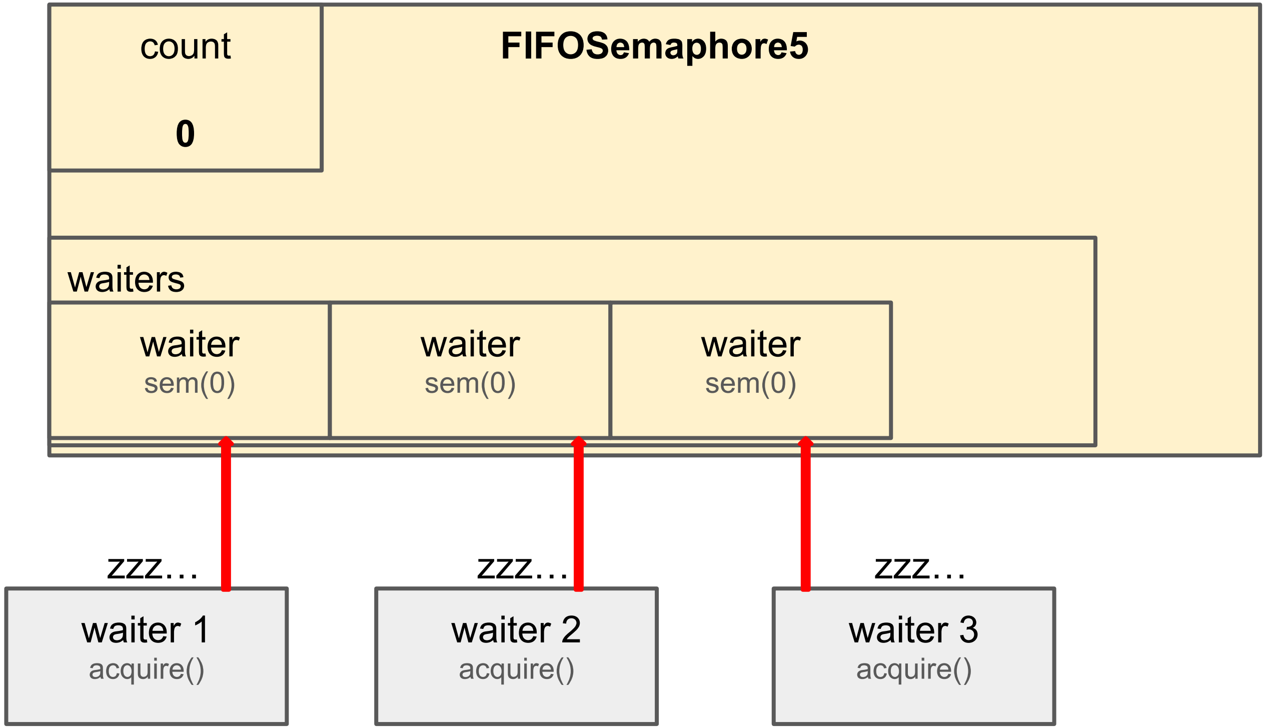
Demo 5: Queue of semaphores

```cpp
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} {}
```

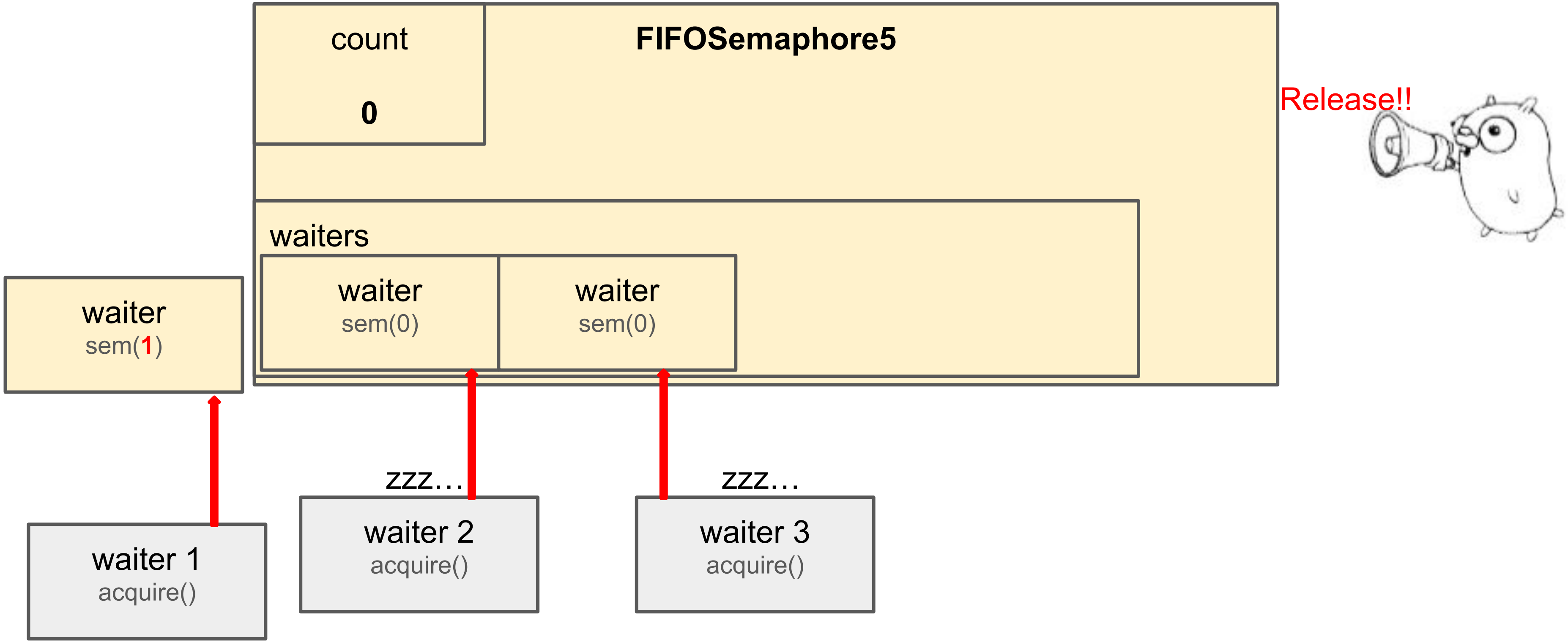
```cpp
void acquire() {
  auto waiter = std::make_shared<Waiter>();
  {
    std::scoped_lock lock{mut};
    if (count > 0) {
      count--;  // Positive count,
      return;   // simply decrement without blocking
    }
    waiters.push(waiter);  // Zero count, add to waiters
  }
  waiter->sem.acquire();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

```cpp
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
}
```

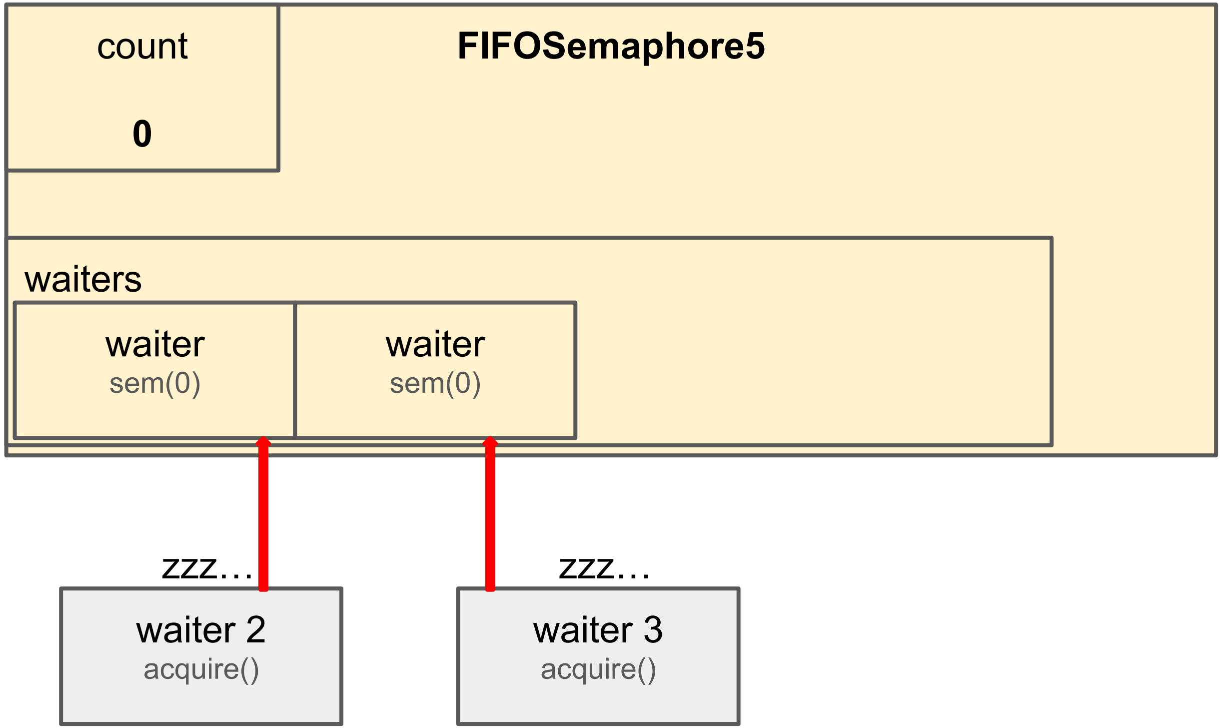
Demo 5: Queue of semaphores



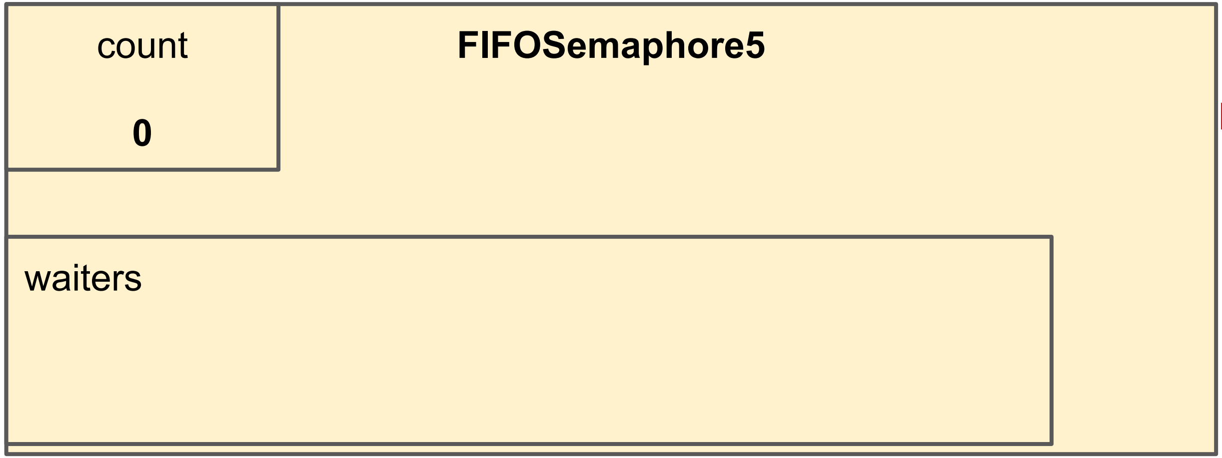
Demo 5: Queue of semaphores



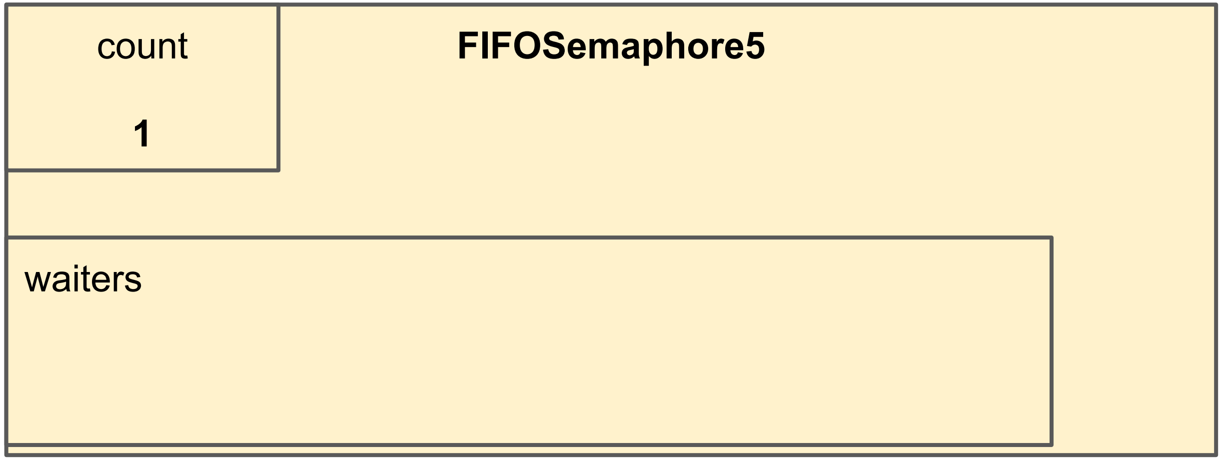
Demo 5: Queue of semaphores



Demo 5: Queue of semaphores



Demo 5: Queue of semaphores



Demo 5: extras

Issues:

- 2 allocations each time waiter is added to queue

How about we do only 1 allocation?

Demo 5: extras

```
void acquire() {
  Waiter waiter;

  {
    std::scoped_lock lock{mut};
    if (count > 0) {
      count--;
      return;
    }

    if (back == nullptr) {
      front = back = &waiter;
    } else {
      back->next = &waiter;
      back = &waiter;
    }
  }
  waiter.sem.acquire();
}
```

Since destruction of waiter is after all “external” accesses to it, no invalid memory access.
tldr; no lifetime issues

```
struct FIFOSemaphore7 {
  struct Waiter {
    std::binary_semaphore sem{0};
    Waiter *next{nullptr};
  };

  std::mutex mut;
  Waiter *front;
  Waiter *back;
  std::ptrdiff_t count;

  FIFOSemaphore7(std::ptrdiff_t initial_count)
    : mut{}, front{nullptr}, back{nullptr}, count{initial_count} {}
```

```
void release() {
  Waiter *head;
  {
    std::scoped_lock lock{mut};
    if (front == nullptr) {
      count++;
      return;
    }
    head = front;
    front = front->next;
    if (front == nullptr) {
      back = nullptr;
    }
  }
  head->sem.release();
}
```

Demo 6: Using a buffered channel

- Sending on a full/unbuffered channel -> block

- Suppose initial_count = N
  - Make a buffered channel of size N.
  - If M > N goroutines arrive at the same time, (M - N) goroutines block.
  - release -> send to channel
  - acquire -> recv from channel

Basically a semaphore!

Demo 6: Using a buffered channel

```
type hchan struct {
        qcount   uint           // total data in the queue
        dataqsiz uint           // size of the circular queue
        buf      unsafe.Pointer // points to an array of dataqsiz elements
        elemsize uint16
        closed   uint32
        elemtype *_type // element type
        sendx    uint   // send index
        recvx    uint   // receive index
        recvq    waitq  // list of recv waiters
        sendq    waitq  // list of send waiters
```

```
type waitq struct {
        first *sudog
        last  *sudog
}
```

```
type sudog struct {
        // The following fields are protected by the hchan.lock of the
        // channel this sudog is blocking on. shrinkstack depends on
        // this for sudogs involved in channel ops.

        g *g

        next *sudog
        prev *sudog
```

current impl is fifo, but not guaranteed by the standard.

Demo 7: Using a daemon goroutine

Bad idea to rely on implementation-defined behaviour.

How about we have a goroutine manage others!

Demo 7: Using a daemon goroutine

```go
type Semaphore2 struct {
    acquireCh chan chan struct{}
    releaseCh chan struct{}
}
```

```go
func NewSemaphore2(initial_count int) *Semaphore2 {
    sem := new(Semaphore2)
    sem.acquireCh = make(chan chan struct{}, 100)
    sem.releaseCh = make(chan struct{}, 100)

    go func() {
        count := initial_count
        // The FIFO queue that stores the channels used to unblock waiters
        waiters := NewChanQueue()
```

```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
}

func (s *Semaphore2) Release() {
    s.releaseCh <- struct{}{}
}
```

Demo 7: Using a daemon goroutine

```go
for {
		select {
		case <-sem.releaseCh:
		   // …

		case ch := <-sem.acquireCh:
		   // …
	}
```

`Either process a release or acquire request`

Demo 7: Using a daemon goroutine

```
        case ch := <-sem.acquireCh: // Decrement or add a waiter
            if count > 0 {
                count--
                ch <- struct{}{} // Since count is +ve, don’t block the waiter
            } else {
                waiters.PushBack(ch) // Add waiter to back of the wait queue
            }
        }
```

Either process a release or acquire request

Demo 7: Using a daemon goroutine

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

Either process a release or acquire request

Demo 7: Using a daemon goroutine

Why a buffer of 100?

```go
func NewSemaphore2(initial_count int) *Semaphore2 {
    sem := new(Semaphore2)
    sem.acquireCh = make(chan chan struct{}, 100)
    sem.releaseCh = make(chan struct{}, 100)

    go func() {
        count := initial_count
        // The FIFO queue that stores the channels used to unblock waiters
        waiters := NewChanQueue()
```

Demo 7: Using a daemon goroutine

Why a buffer of 100?

If goroutine blocks on

```
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
}
```

for a long time (wakeups not in FIFO), no guarantees of FIFO.

BUT, if send succeeds (buffered in channel), we get FIFO behavior.

```
type hchan struct {
        qcount   uint           // total data in the queue
        dataqsiz uint           // size of the circular queue
        buf      unsafe.Pointer // points to an array of dataqsiz elements
        elemsize uint16
        closed   uint32
        elemtype *_type // element type
        sendx    uint   // send index
        recvx    uint   // receive index
        recvq    waitq  // list of recv waiters
        sendq    waitq  // list of send waiters

        // lock protects all fields in hchan, as well as several
        // fields in sudogs blocked on this channel.
        //
        // Do not change another G's status while holding this lock
        // (in particular, do not ready a G), as this can deadlock
        // with stack shrinking.
        lock mutex
}
```

Pls scan for attendance

See you next week!