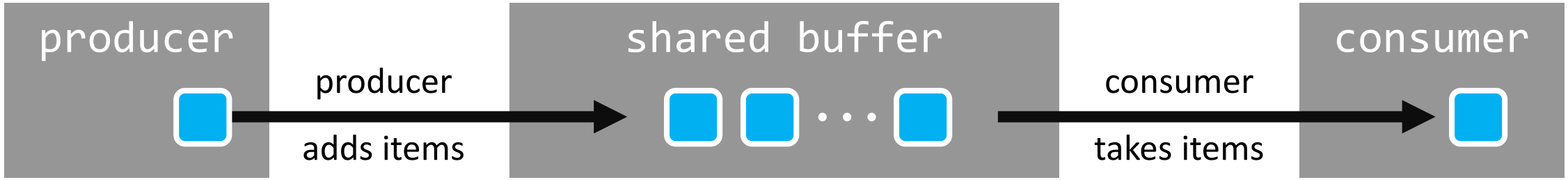
Yesterday on PP:

Synchronized, Wait, Notify, NotifyAll, Deadlocks, Lockouts, Nondeterminism, …

Producer-Consumer

Producer and consumer run indefinitely

Producer puts items into a shared buffer, consumer takes them out



For simplicity, buffer is unbounded (has no capacity limit); producing is always possible

But consumption only possible if buffer isn't empty

Producer-Consumer: v3

```
public class Consumer extends Thread {
  ...

  public void run() {
    long prime;
    while (true) {
      synchronize (buffer) {
        while (buffer.isEmpty())
          buffer.wait();
        prime = buffer.remove();
      }
      performLongRunningComputation(prime);
    }
  }
}
```

```
public class Producer extends Thread {
  ...

  public void run() {
    ...

    while (true) {
      prime = computeNextPrime(prime);
      synchronize (buffer) {
        buffer.add(prime);
        buffer.notifyAll();
      }
    }
  }
}
```

`buffer.wait()`:

1. Consumer thread goes to sleep (status NOT RUNNABLE) …
2. … and gives up buffer's lock

`buffer.notifyAll()`:

1. All threads waiting for buffer's lock are woken up (status RUNNABLE)

Pseudo-Code Implementations (Inside JVM)

```
synchronized(obj) { s } ≡
  obj.acquireLock();
  s;
  obj.releaseLock();
```

```
obj.acquireLock() ≡
  label L:
  if (obj.owner == null) {
    obj.owner == currentThread;
  } else {
    currentThread.sleepUntilLockReleasedFor(obj);
    // Next line executed only once thread woken up
    goto L;
  }
```

```
obj.releaseLock() ≡
  assert obj.owner == currentThread;
  obj.owner = null
  informThreadsWaitingOn(obj);
```

Pseudo-code implementations providing intuition for how these operations could be implemented inside the JVM.

We assume that bad interleavings can't happen inside the pseudo-code, e.g

```
  if (obj.owner == null)
```

and subsequent statement

```
  obj.owner == currentThread;
```

aren't interrupted by another thread (the JVM indeed takes care of this).

Details such as the bookkeeping of which thread wants to acquire which lock are omitted here (and not necessary for building intuition). However, they will be provided in the 2nd half of the course.

Pseudo-Code Implementations Inside JVM

```
obj.wait() ≡
   obj.releaseLock();
   label L:
   currentThread.sleepUntilNotifiedOn(obj);
   // Next line executed only once thread woken up
   obj.acquireLock();
```

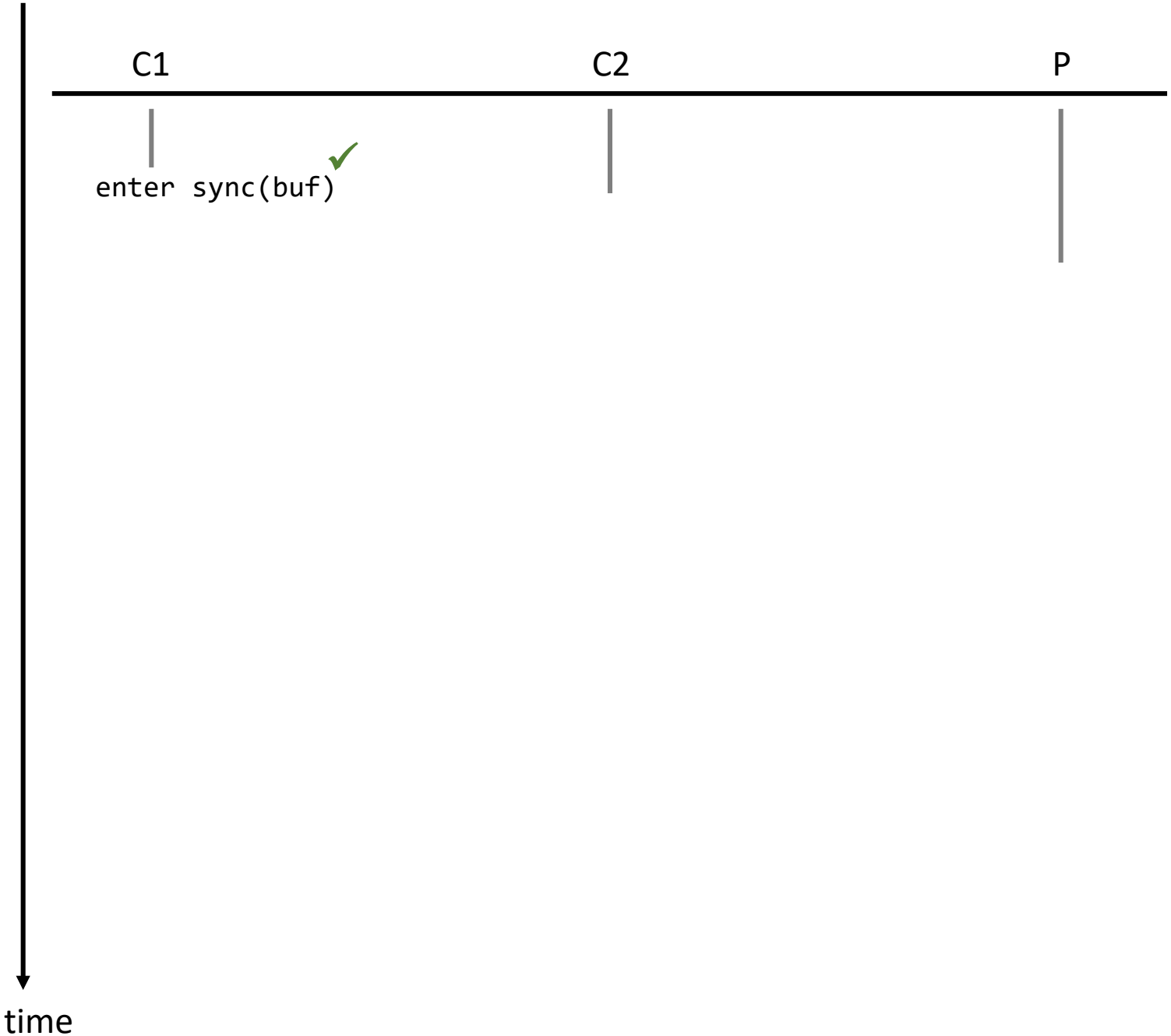
```
obj.notify() ≡
   informSomeThreadWaitingOn(obj)
```

```
obj.notifyAll() ≡
   informAllThreadsWaitingOn(obj)
```

Pseudo-code implementations providing intuition for how these operations could be implemented inside the JVM.

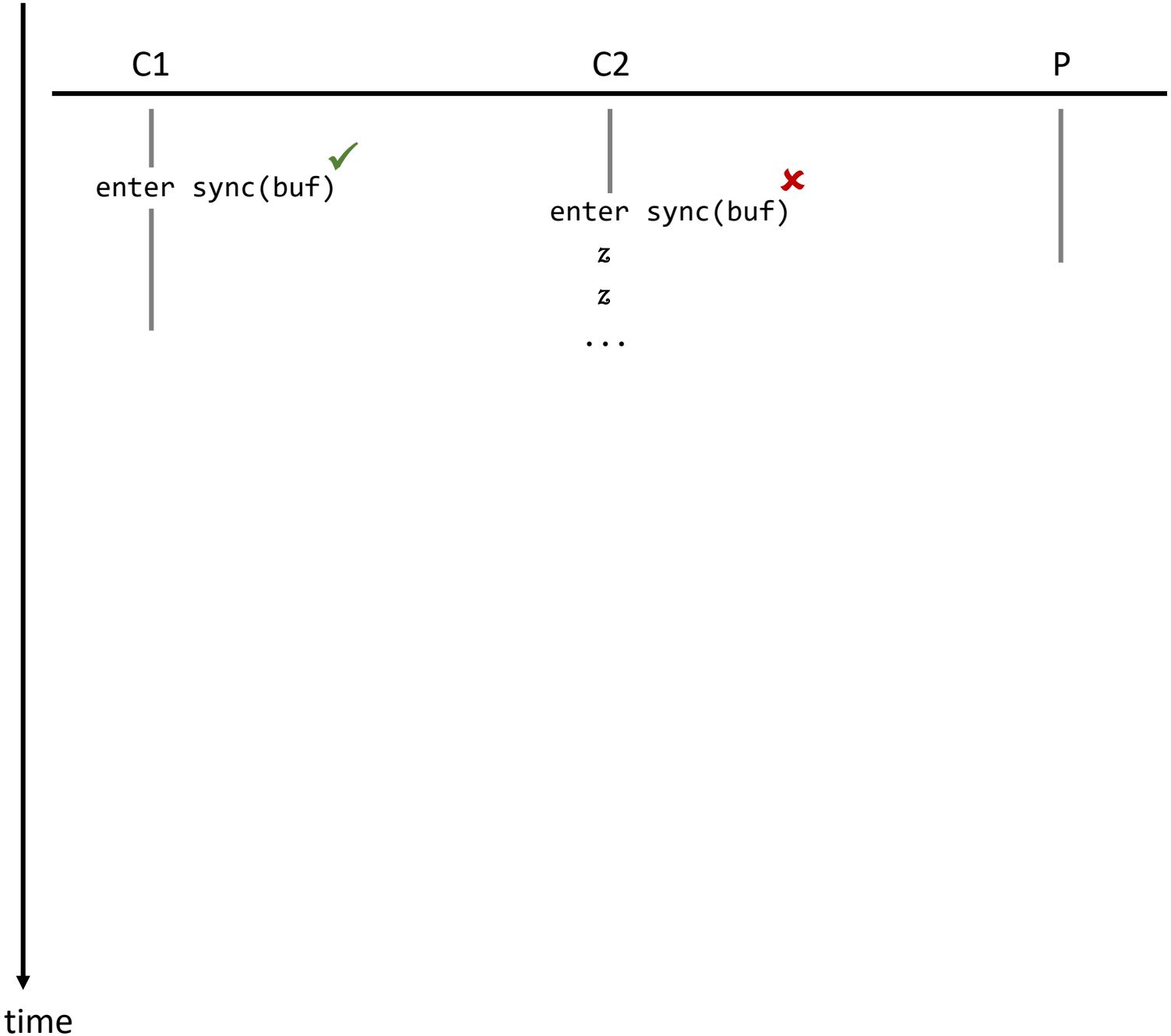
We assume that bad interleavings can't happen inside the pseudo-code.

Details such as the bookkeeping of which thread wants to acquire which lock are omitted here (and not necessary for building intuition). However, they will be provided in the 2nd half of the course.



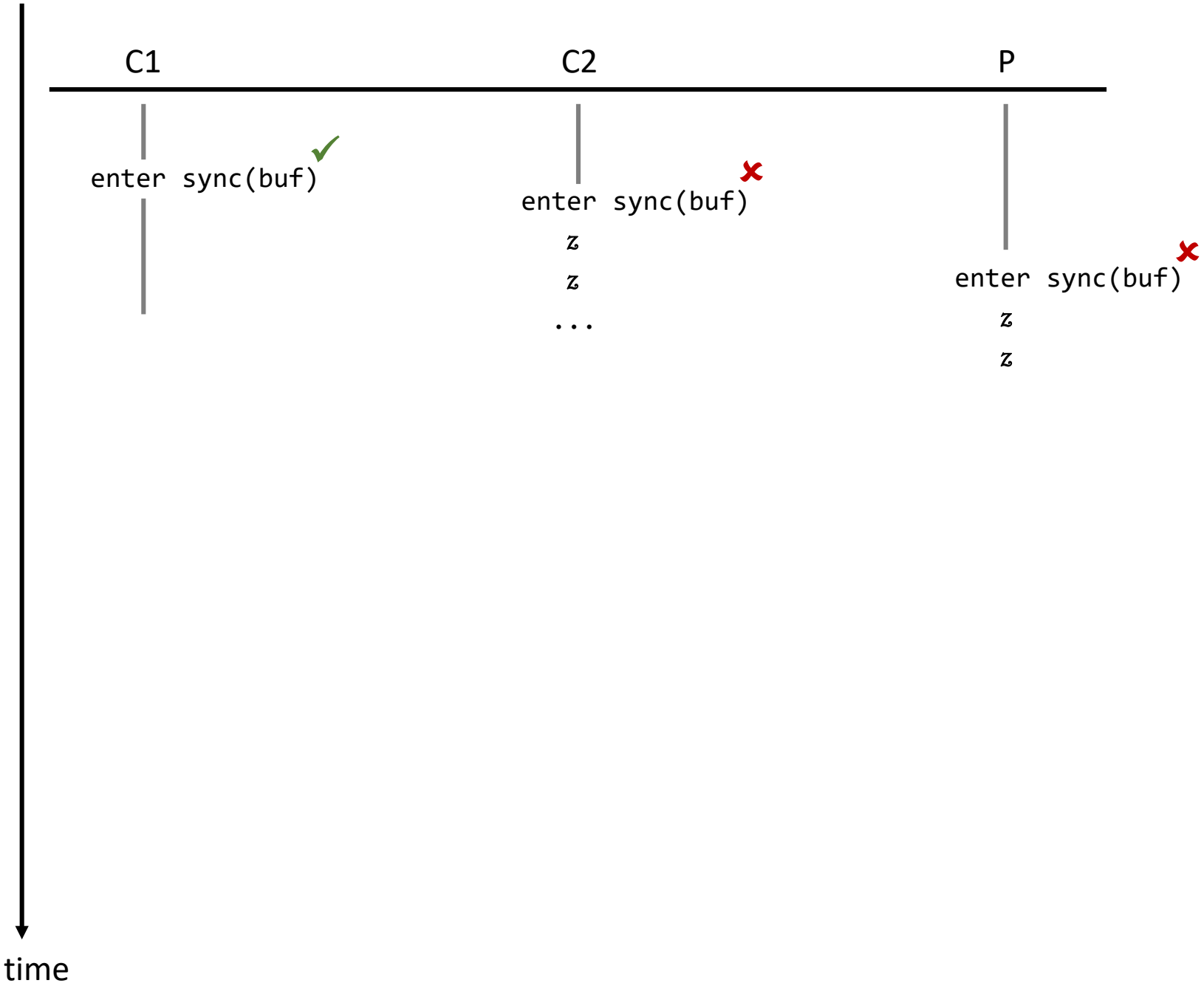
Buffer
owner:
…

Use the pseudo-code on the previous slides to explore this and other scenarios.



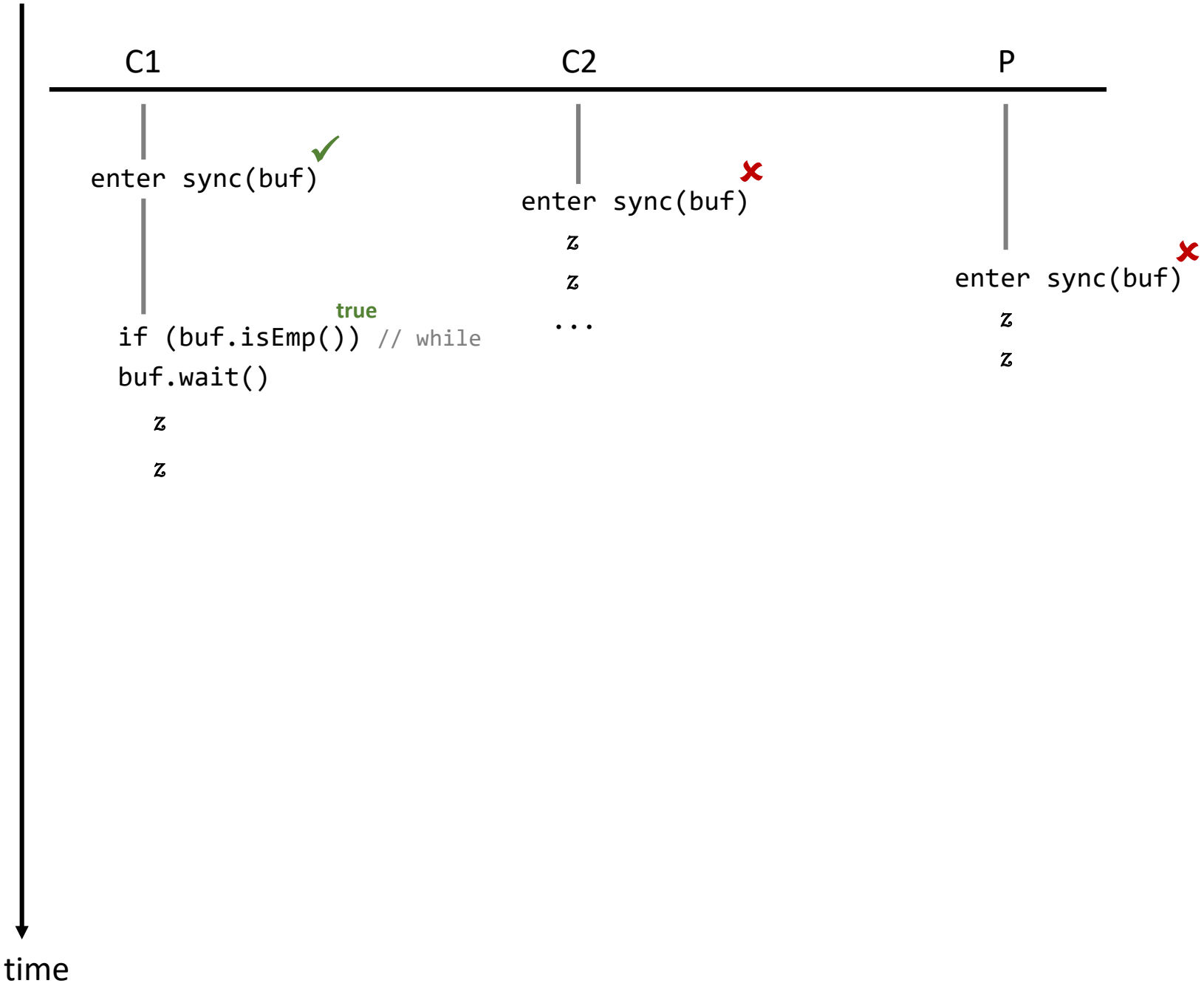
Buffer
owner: C1
...

Use the pseudo-code on the previous slides to explore this and other scenarios.



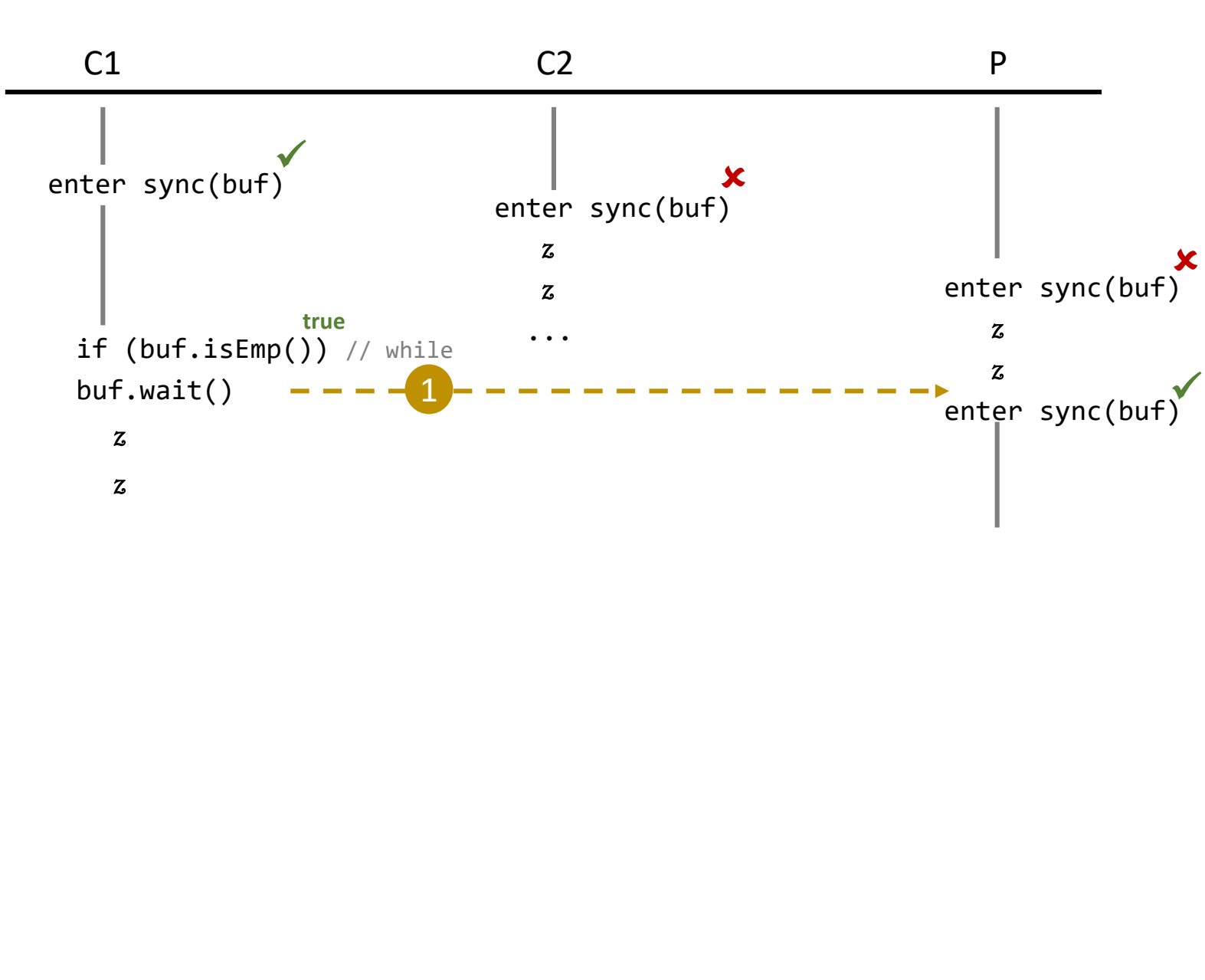
Buffer
owner: C1
...

Use the pseudo-code on the previous slides to explore this and other scenarios.



Buffer
owner: ~~C1~~
...

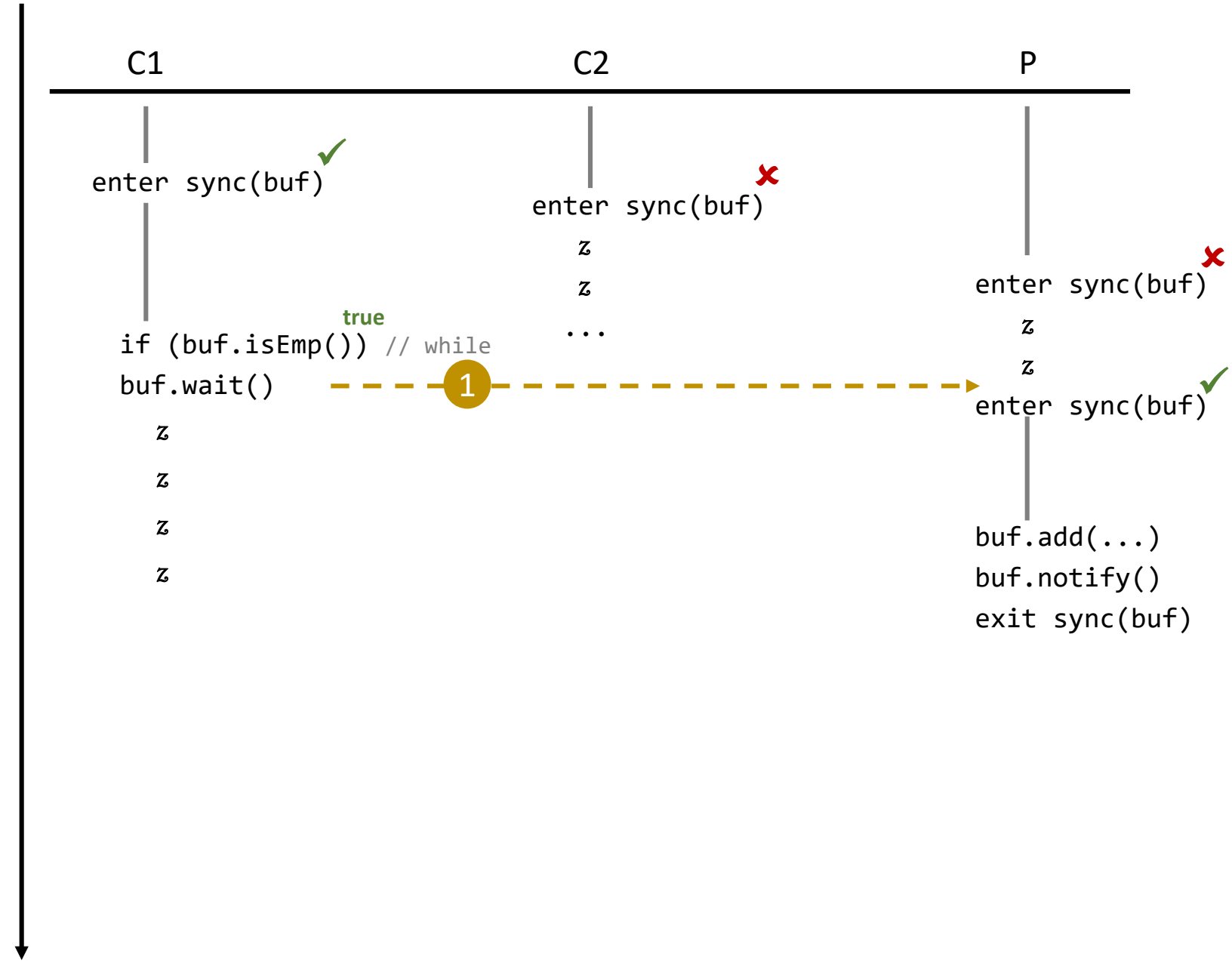
Use the pseudo-code on the previous slides to explore this and other scenarios.



Buffer
owner: ~~C1~~, P
…

1 buf.wait() release buf's lock, other threads are woken up, P gets buf's lock
(C2 could also get it – what would happen then? spoiler: nothing bad)

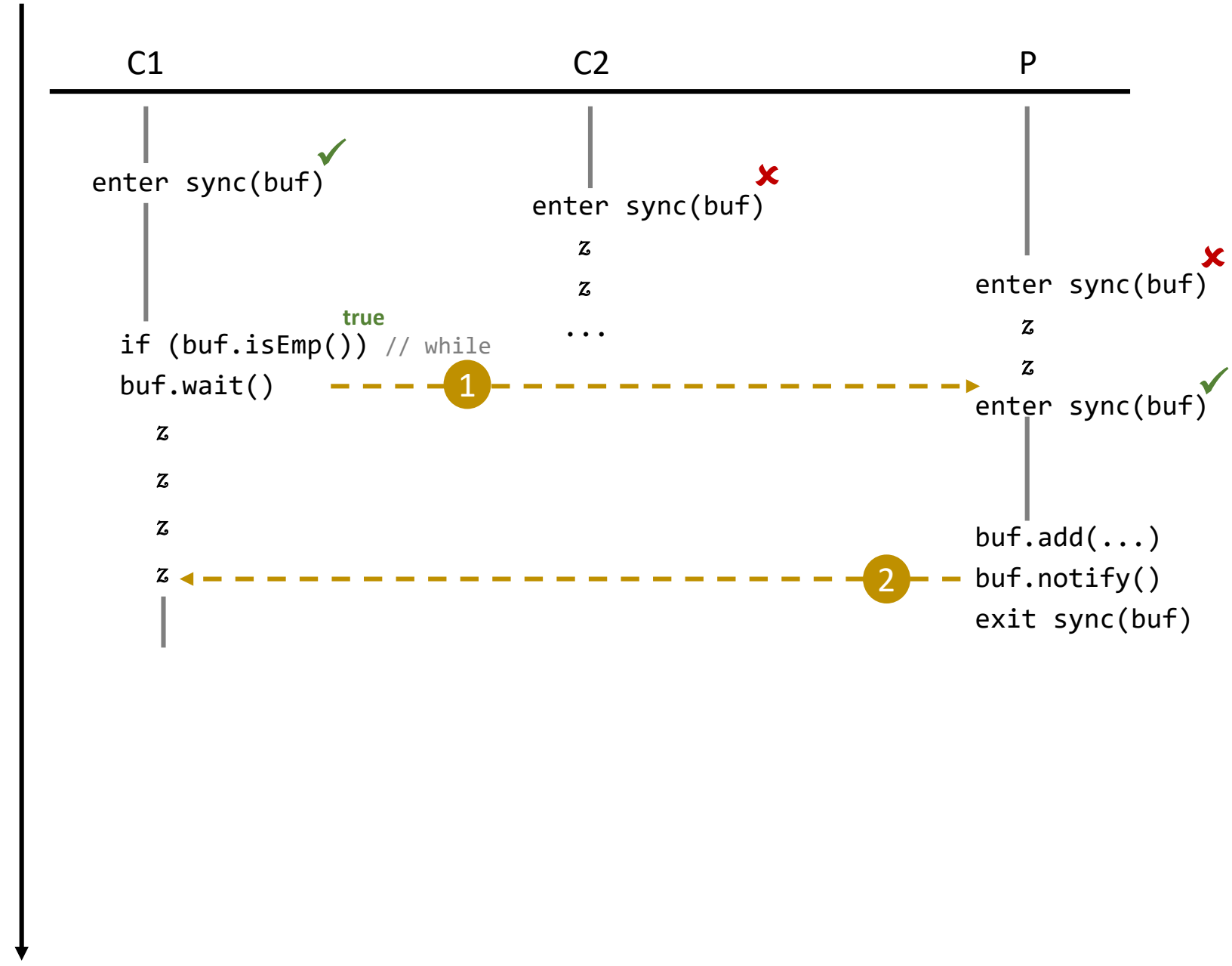
Use the pseudo-code on the previous slides to explore this and other scenarios.



Buffer
owner: ~~C1~~, P
...

1 buf.wait() release buf's lock, other threads are woken up, P gets buf's lock
(C2 could also get it – what would happen then? spoiler: nothing bad)

Use the pseudo-code on the previous slides to explore this and other scenarios.

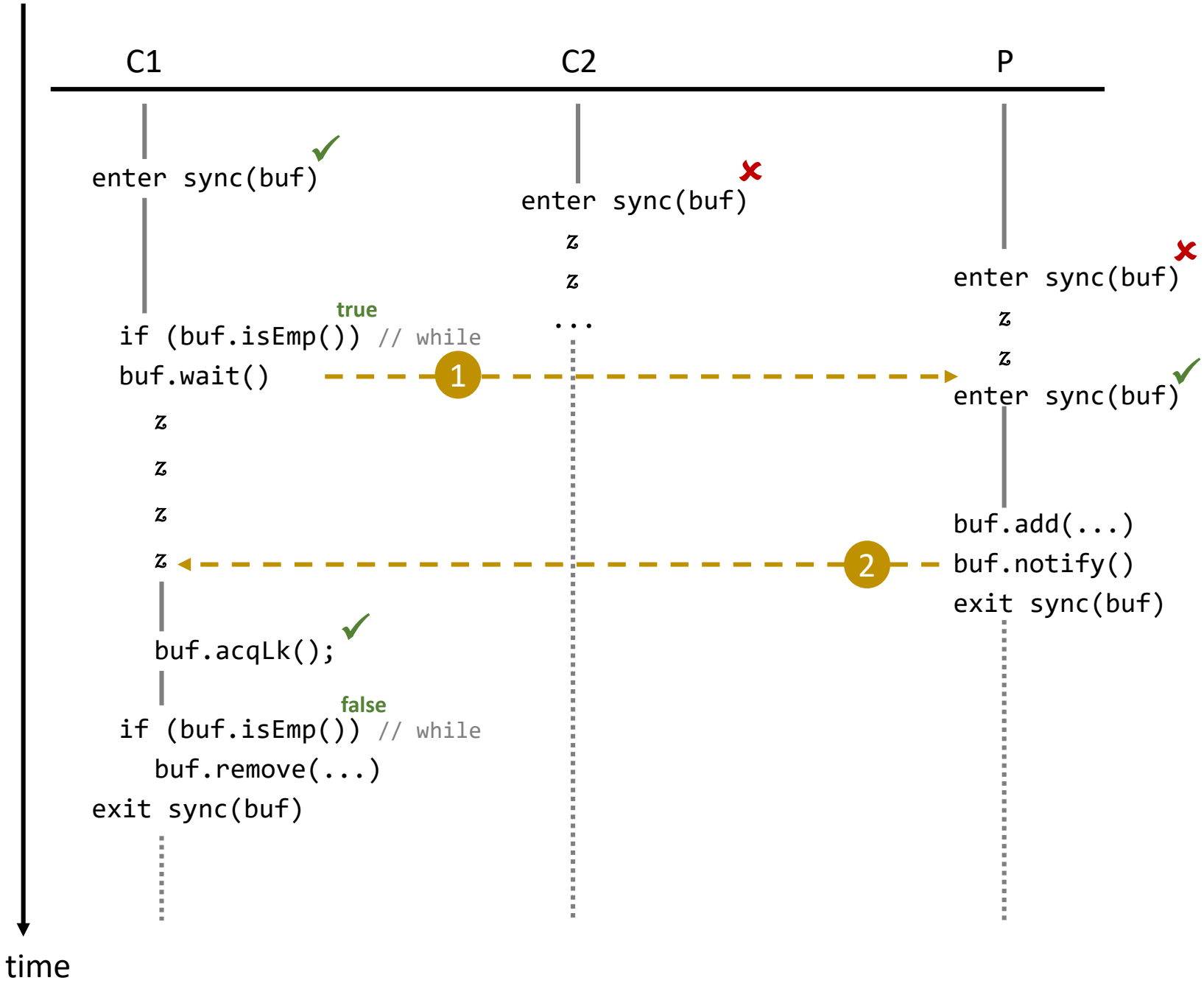


Buffer
owner: ~~C1~~, ~~P~~
...

1. buf.wait() release buf's lock, other threads are woken up, P gets buf's lock
(C2 could also get it – what would happen then? spoiler: nothing bad)

2. buf.notify() informs C1 that the buffer has been modified
(what if C1 immediately tries to acquire buf's lock? I.e. before P releases it? Or what if C2 again attempts to get buf's lock? spoiler: nothing bad)

Use the pseudo-code on the previous slides to explore this and other scenarios.



Buffer
owner: ~~C1~~, ~~P~~, C1
...

1. buf.wait() release buf's lock, other threads are woken up, P gets buf's lock
(C2 could also get it – what would happen then? spoiler: nothing bad)

2. buf.notify() informs C1 that the buffer has been modified
(what if C1 immediately tries to acquire buf's lock? I.e. before P releases it? Or what if C2 again attempts to get buf's lock? spoiler: nothing bad)

Use the pseudo-code on the previous slides to explore this and other scenarios.

Don’t worry, all of this will be elaborated on in the 2nd half