Intro to Semaphores

CS 492 Operating Systems

Guest Lecturer: Ryan Tsang

Handout: github.com/rctsang/sit-cs492-semaphores

Learning Objectives

- Define a **semaphore** and its operations
- Reason about multi-process execution with semaphores
- Understand the role of semaphores as a **resource counter**
- Use semaphores to resolve the **producer-consumer problem**
- Define the **binary semaphore**
- Recognize impact of **order of operations**

Roadmap

- Warm-Up and Recap
- Semaphores
- P-C Problem with Semaphores?
- Binary Semaphores
- Revisiting P-C Problem with Semaphores

Warm-Up and Recap

producer-consumer problem

Producer-Consumer Problem

```
1:  /** producer-consumer with race condition */
2:
3:  #define N 3      // number of slots in buffer
4:  int count = 0;   // number of items in buffer
5:
6:  void producer(void)
7:  {
8:      int item;
9:      while (1) {
10:         item = produce_item();
11:         if (count == N) {
12:             sleep();
13:         }
14:         insert_item(item);
15:         count++;
16:         if (count == 1) {
17:             wakeup(consumer);
18:         }
19:     }
20: }
```

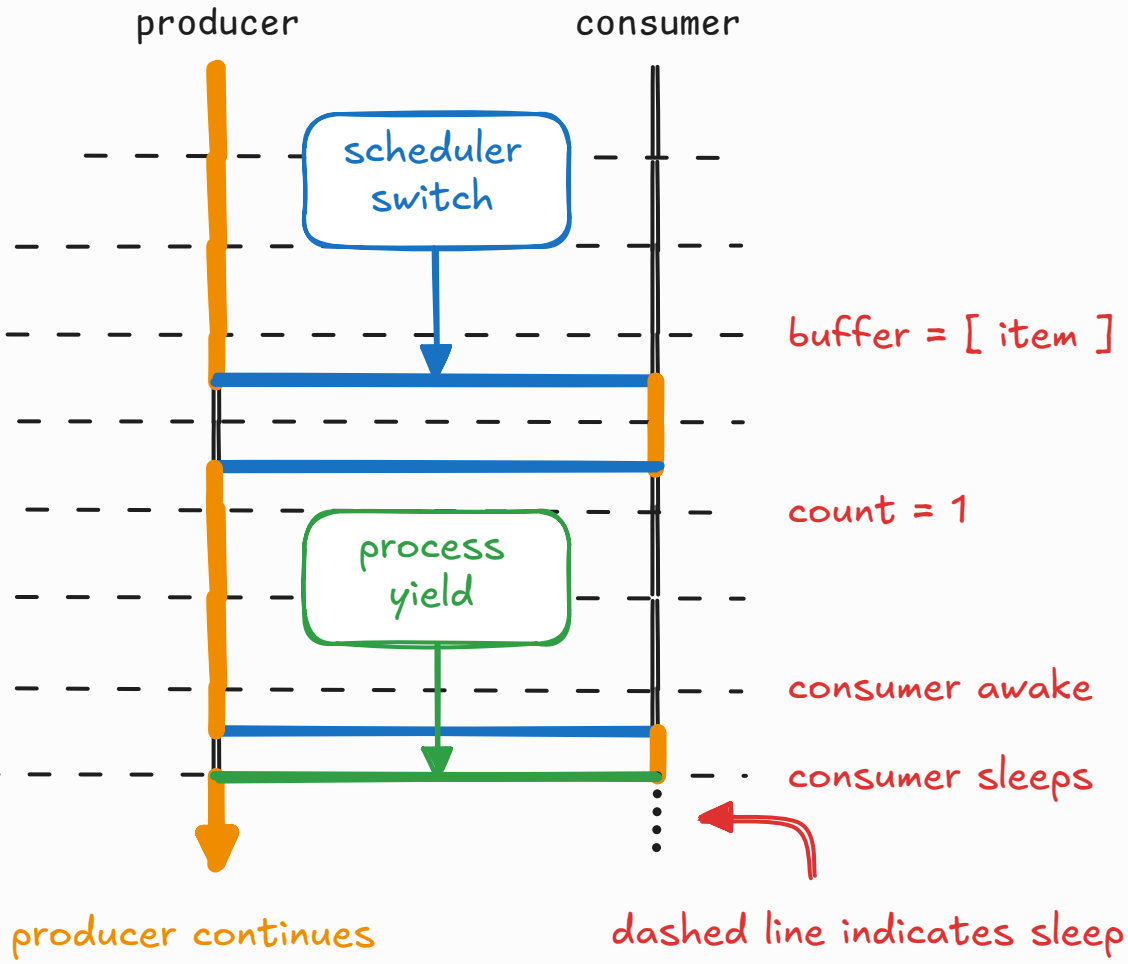
```
21: void consumer(void)
22: {
23:     int item;
24:     while (1) {
25:         if (count == 0) {
26:             sleep();
27:         }
28:         item = remove_item();
29:         count--;
30:         if (count == N - 1) {
31:             wakeup(producer);
32:         }
33:         consume_item(item);
34:     }
35: }
```

What sequence of events will cause both processes to sleep forever?

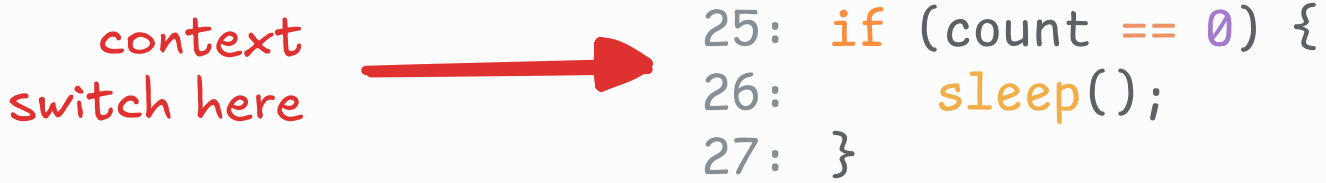
What's the issue?

```
init:
  buffer = [ ]
  count = 0

producer 10: item = produce_item()
producer 11: if (count == N)
producer 14: insert_item(item)
consumer 25: if (count == 0)
producer 15: count++
producer 16: if (count == 1)
producer 17: wakeup(consumer)
consumer 26: sleep()
```



What's the issue?



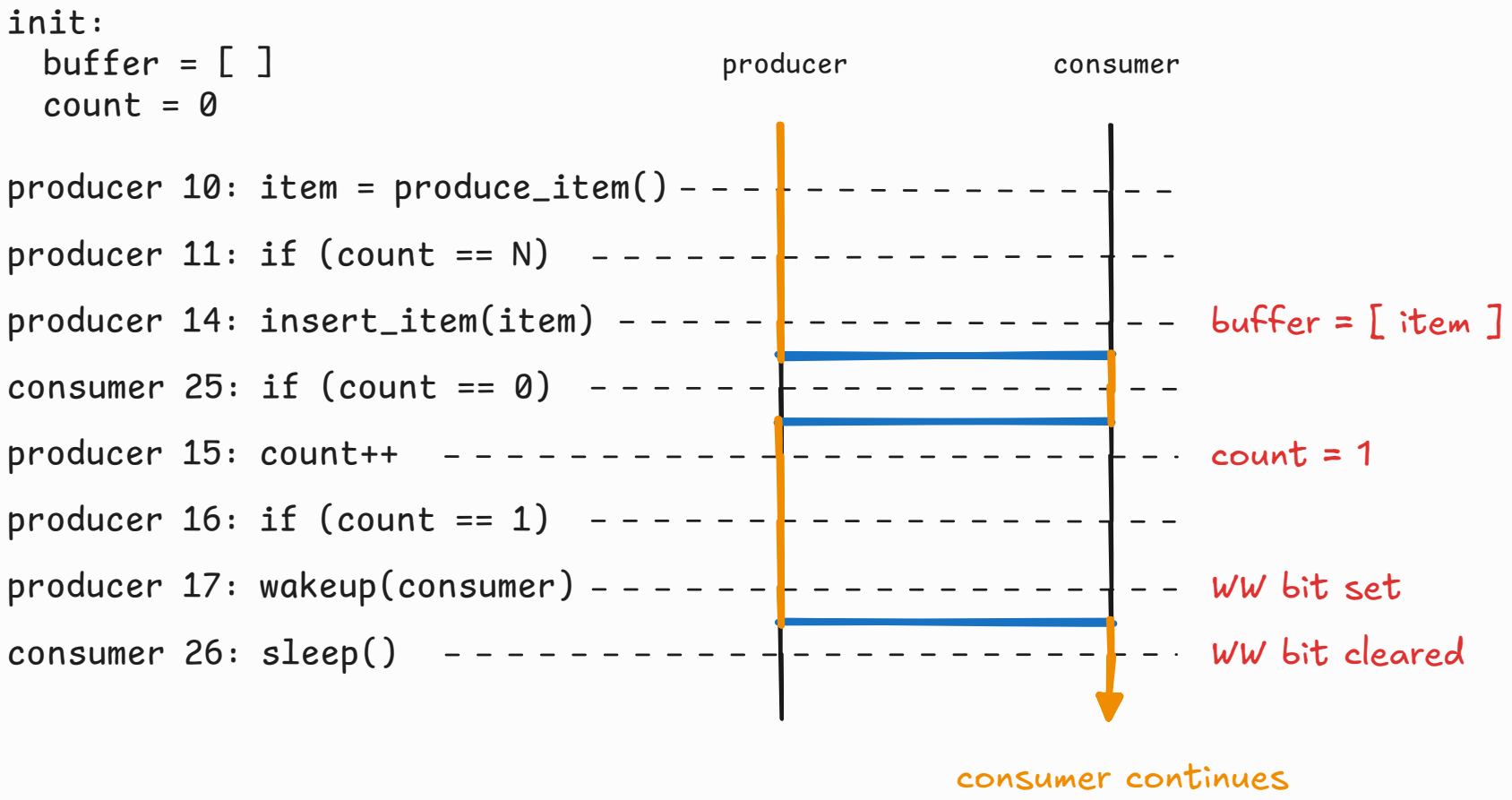
- race condition on the `count` variable.
  - time-of-check time-of-use bug

- wakeup signal is ignored if processes are already awake!

Wakeup-Waiting Bit

- How does the wakeup waiting bit resolve the problem?

The WW bit saves the wakeup signal.



Semaphores

shared resource counters

Semaphores

- A **semaphore** is a shared data type that acts as a *shared counter variable*
  - Counts number of available shared resources
  - Counts number of events, such as pending wakeup signals (like wakeup-waiting bit)
- 2 allowed operations:

down	decrements the semaphore, or suspends execution if the semaphore is already at 0
up	increments the semaphore, or wakes a suspended process if one is present

Pseudocode Implementation

```
atomic down(int * semaphore)
  if semaphore == 0:
    sleep
  else:
    semaphore--
  endif
  return
```

no interrupts while function executing

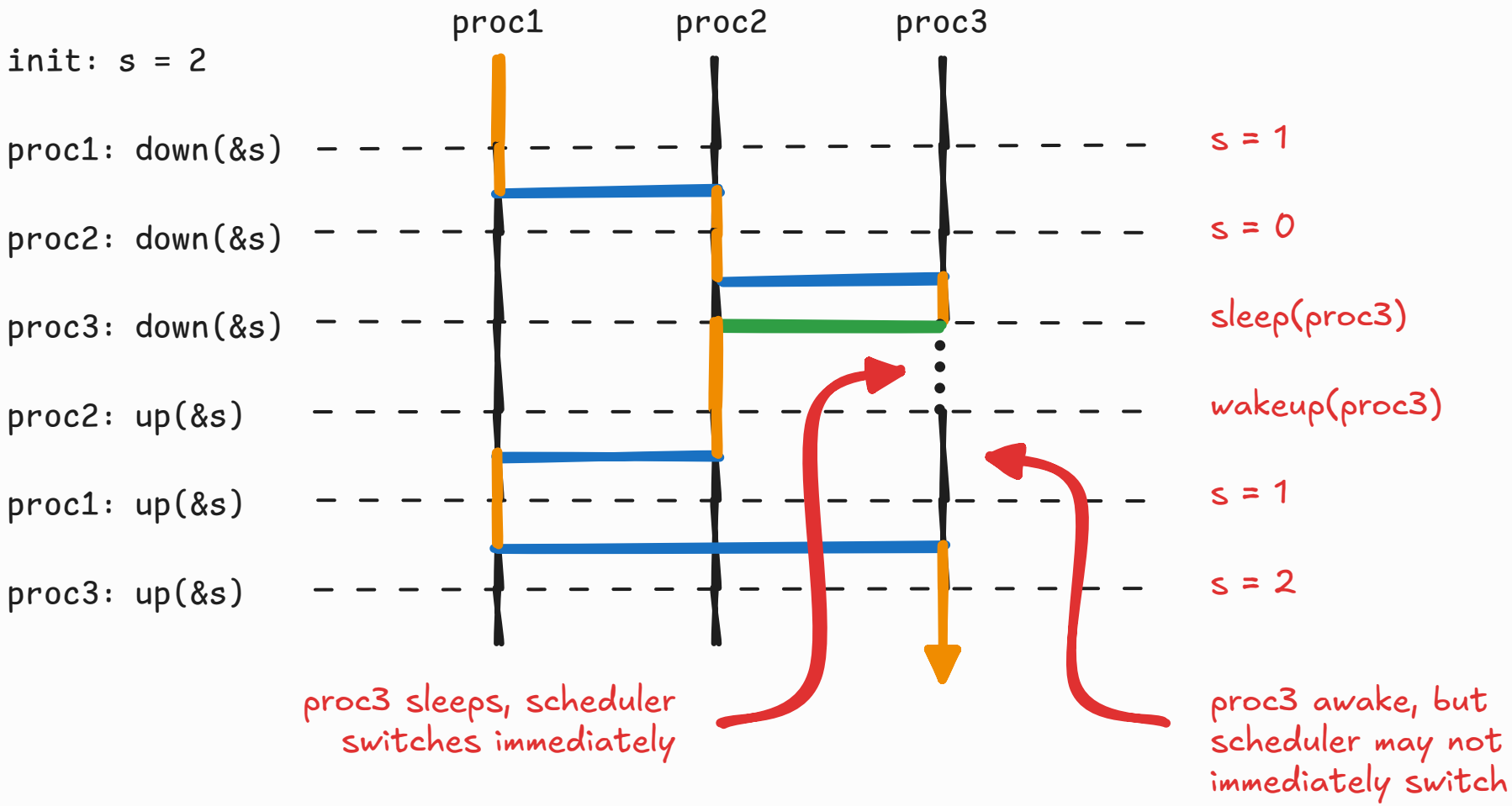
```
atomic up(int * semaphore)
  if semaphore == 0 and has_sleeping(semaphore):
    wake one sleeping process
    return
  else:
    semaphore++
  endif
  return
```

Note 1: actual semaphore implementations vary based on kernel API

Note 2: semaphores often implemented as structs/objects instead of shared integers

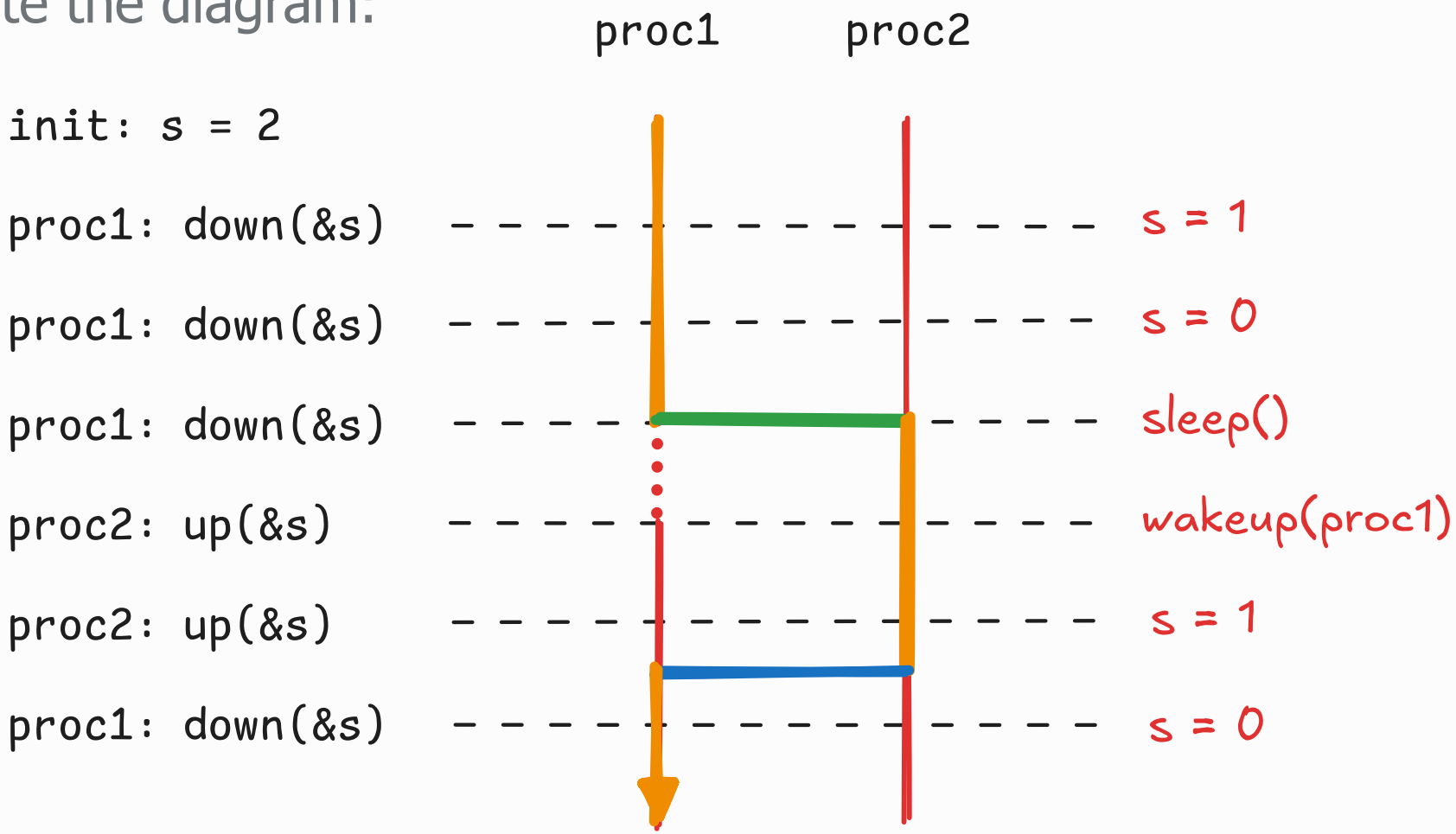
How does this work?

How does this work?



Practice

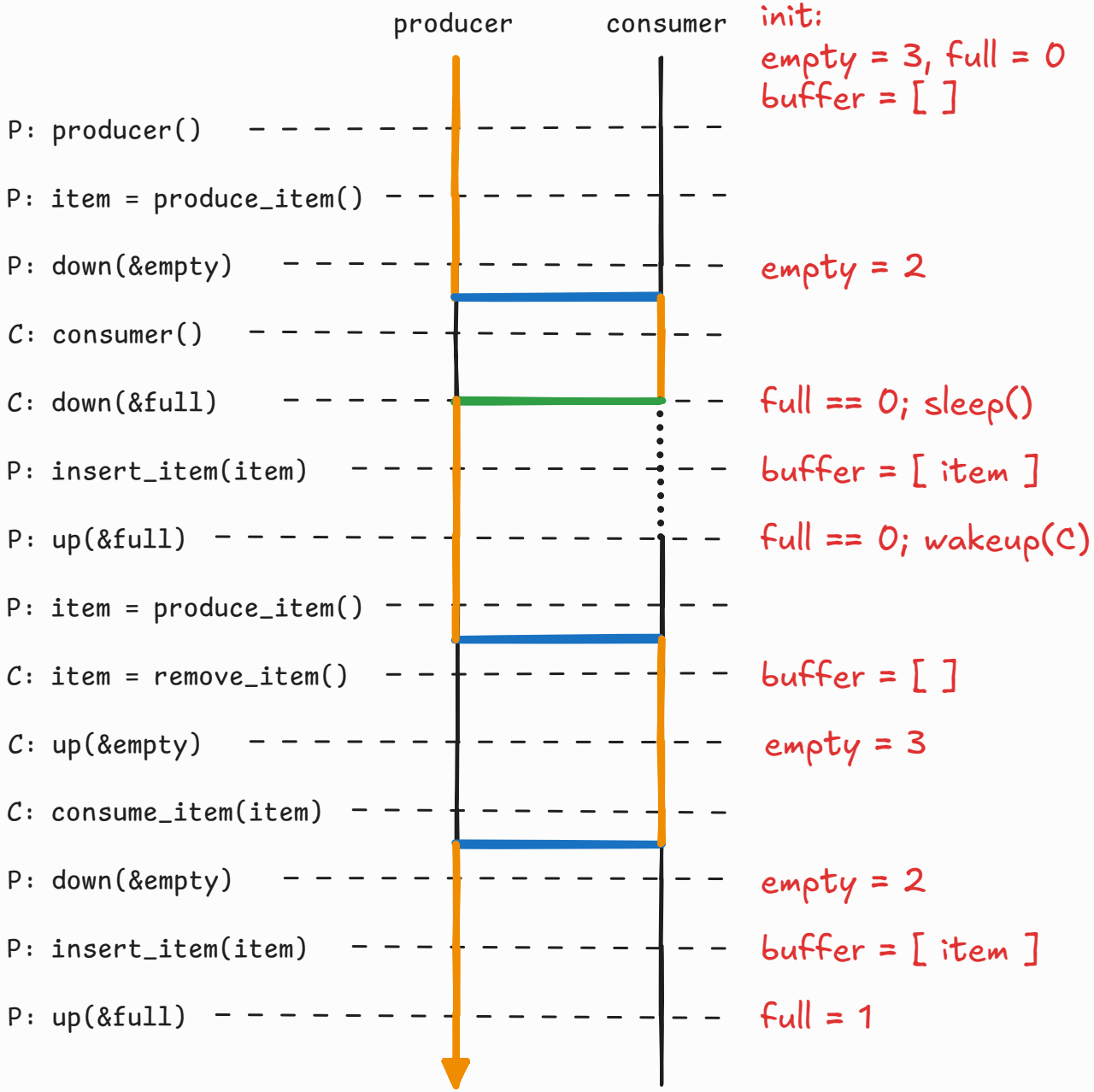
- Complete the diagram:



Producer-Consumer Problem with Semaphores?

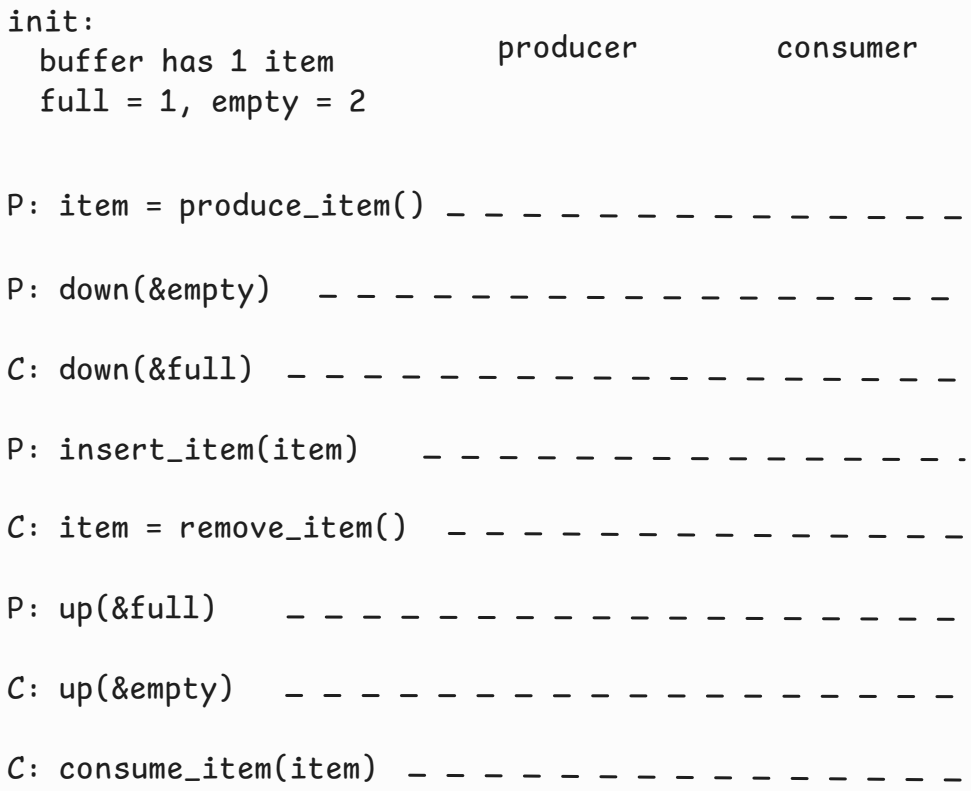
replacing the `count` variable

```
1:  /** producer-consumer with semaphores? */
2:
3:  #define N  3 // number of slots in buffer
4:  typedef int semaphore;
5:  semaphore full  = 0; // count full slots
6:  semaphore empty = N; // count empty slots
7:
8:  void producer(void)
9:  {
10:     int item;
11:     while (1) {
12:         item = produce_item();
13:         down(&empty);
14:         insert_item(item);
15:         up(&full);
16:     }
17: }
18:
19: void consumer(void)
20: {
21:     int item;
22:     while (1) {
23:         down(&full);
24:         item = remove_item();
25:         up(&empty);
26:         consume_item(item);
27:     }
28: }
```



Is the problem solved?

Consider the following scenario:



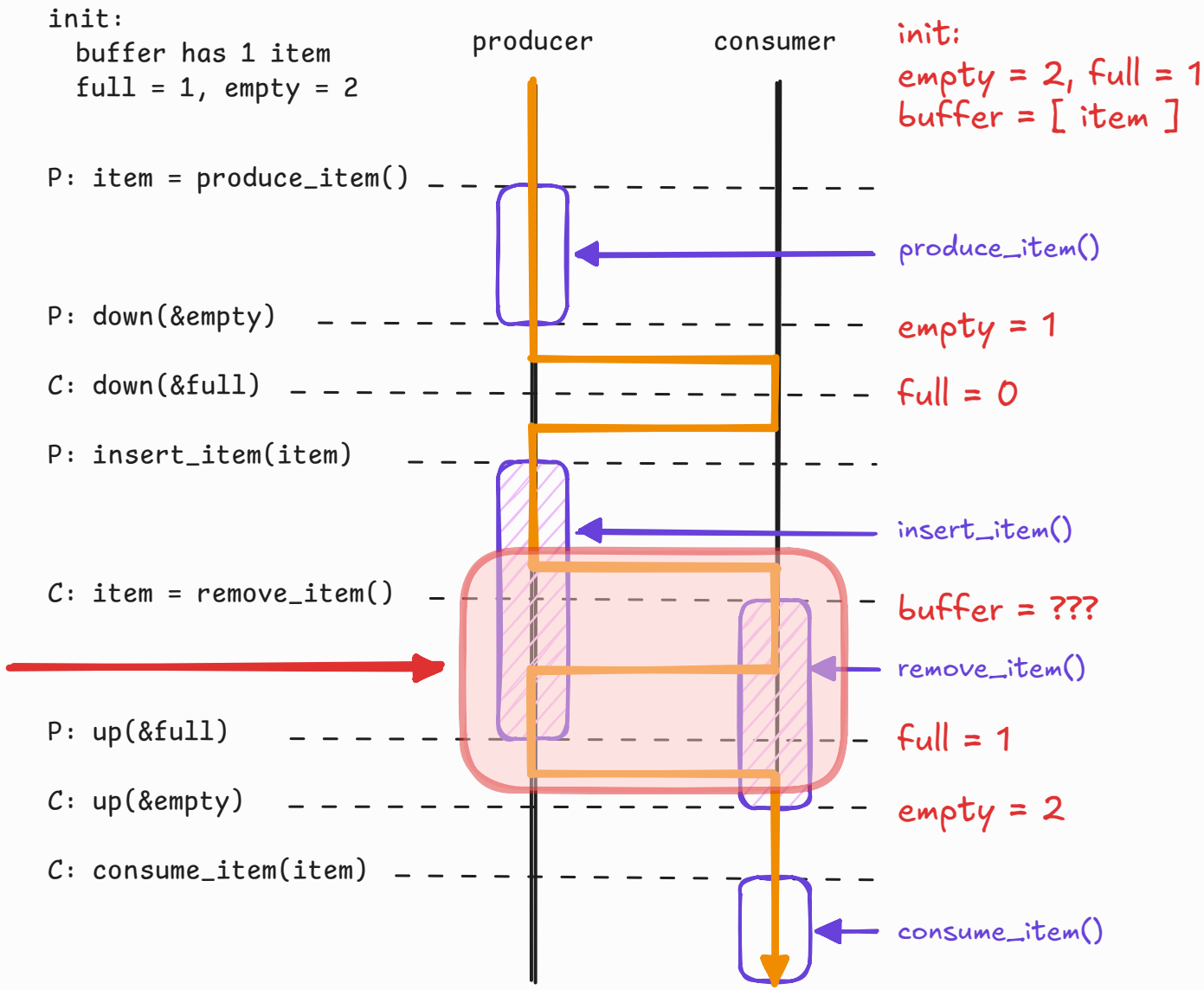
Is the problem solved?

Consider the following scenario:

Remember that insert_item() and remove_item() are still functions!

insert and remove overlapping!
simultaneous access to buffer
is still a race condition

We still need mutual exclusion when accessing the buffer!



Binary Semaphores

counting 1 shared resource

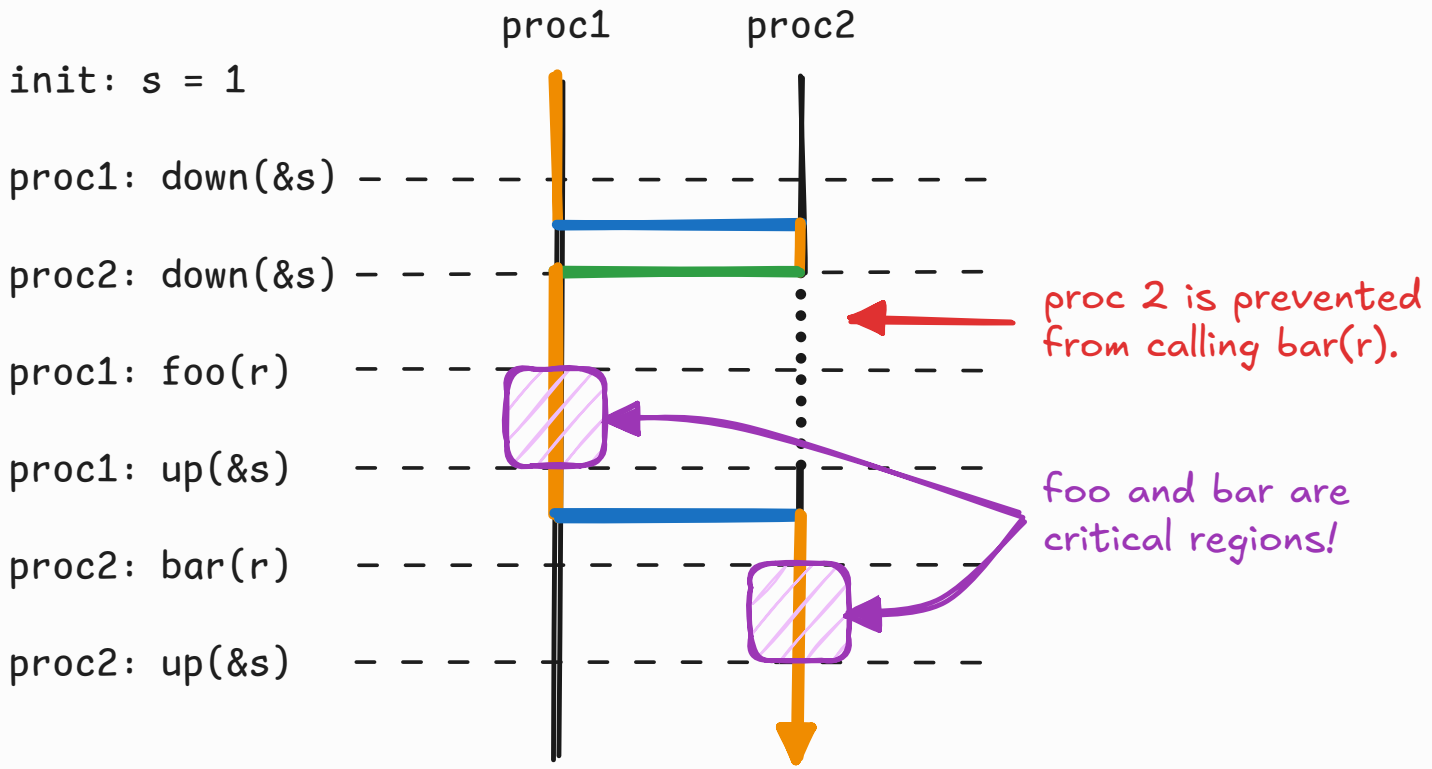
Binary Semaphores

- A **binary semaphore** is a semaphore whose only allowed values are **0** and **1**.
  - In other words, a semaphore for a *single resource*
- Consider the following:

```
1:   semaphore s = 1;
2:   void * r;
3:
4:   void proc1()
5:   {
6:       while(1) {
7:           down(&s);
8:           foo(r);
9:           up(&s);
10:      }
11:  }
12:
```

```
13:  void proc2()
14:  {
15:      while(1) {
16:          down(&s);
17:          bar(r);
18:          up(&s);
19:      }
20:  }
```

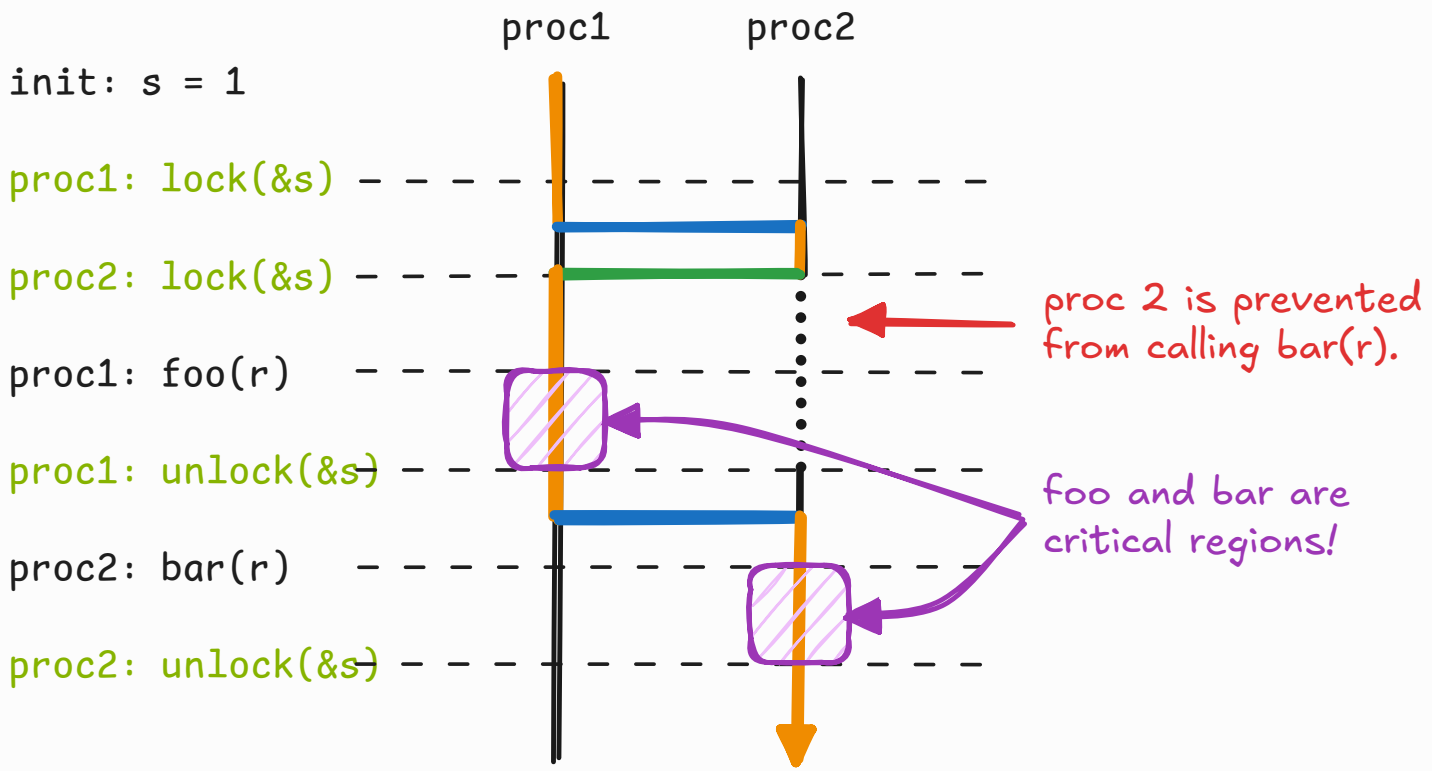
Fill in the following sequence diagram:



A binary semaphore is a mutex lock!

```
 1:  semaphore s = 1;
 2:  void * r;
 3:
 4:  void proc1()
 5:  {
 6:      while(1) {
 7:          down(&s);
 8:          foo(r);
 9:          up(&s);
10:      }
11:  }
12:
13:  void proc2()
14:  {
15:      while(1) {
16:          down(&s);
17:          bar(r);
18:          up(&s);
19:      }
20:  }
```

Fill in the following sequence diagram:



A binary semaphore is a mutex lock!

```
1:  mutex s = 1;
2:  void * r;
3:
4:  void proc1()
5:  {
6:      while(1) {
7:          lock(&s);
8:          foo(r);
9:          unlock(&s);
10:     }
11: }
12:
13: void proc2()
14: {
15:     while(1) {
16:         lock(&s);
17:         bar(r);
18:         unlock(&s);
19:     }
20: }
```

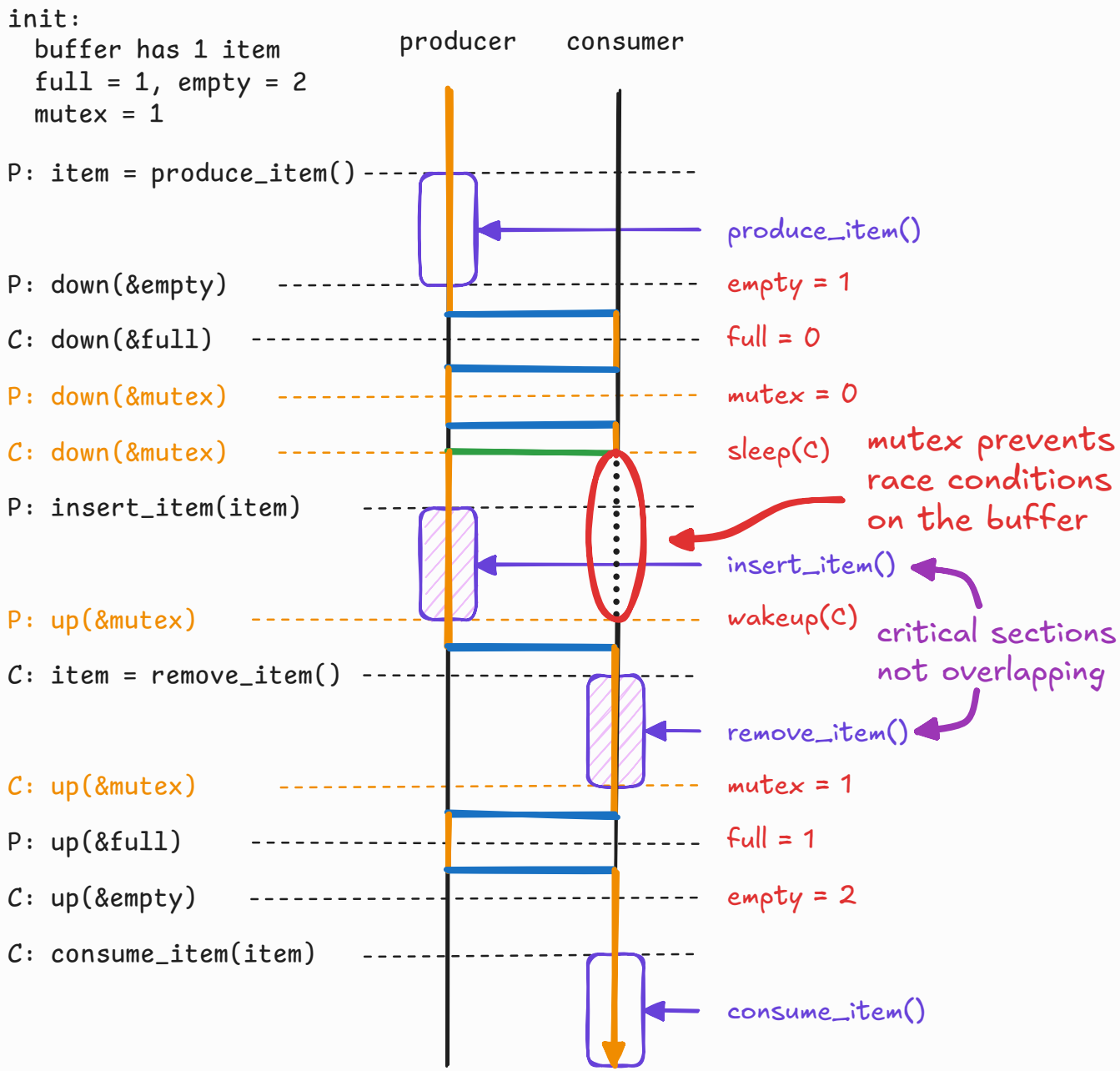
Revisiting Producer-Consumer Problem with Semaphores

for real this time

```
1:  /** producer-consumer with 2 semaphores and a mutex */
2:
3:  #define N 3      // number of slots in buffer
4:  typedef int semaphore;
5:  semaphore full  = 0; // count full slots
6:  semaphore empty = N; // count empty slots
7:  semaphore mutex = 1; // lock buffer access
8:
9:  void producer(void) {
10:     int item;
11:     while (1) {
12:         item = produce_item();
13:         down(&empty);
14:         down(&mutex);
15:         insert_item(item);
16:         up(&mutex);
17:         up(&full);
18:     }
19: }
20:
21: void consumer(void) {
22:     int item;
23:     while (1) {
24:         down(&full);
25:         down(&mutex);
26:         item = remove_item();
27:         up(&mutex);
28:         up(&empty);
29:         consume_item(item);
30:     }
31: }
```

add a binary semaphore to sync buffer access

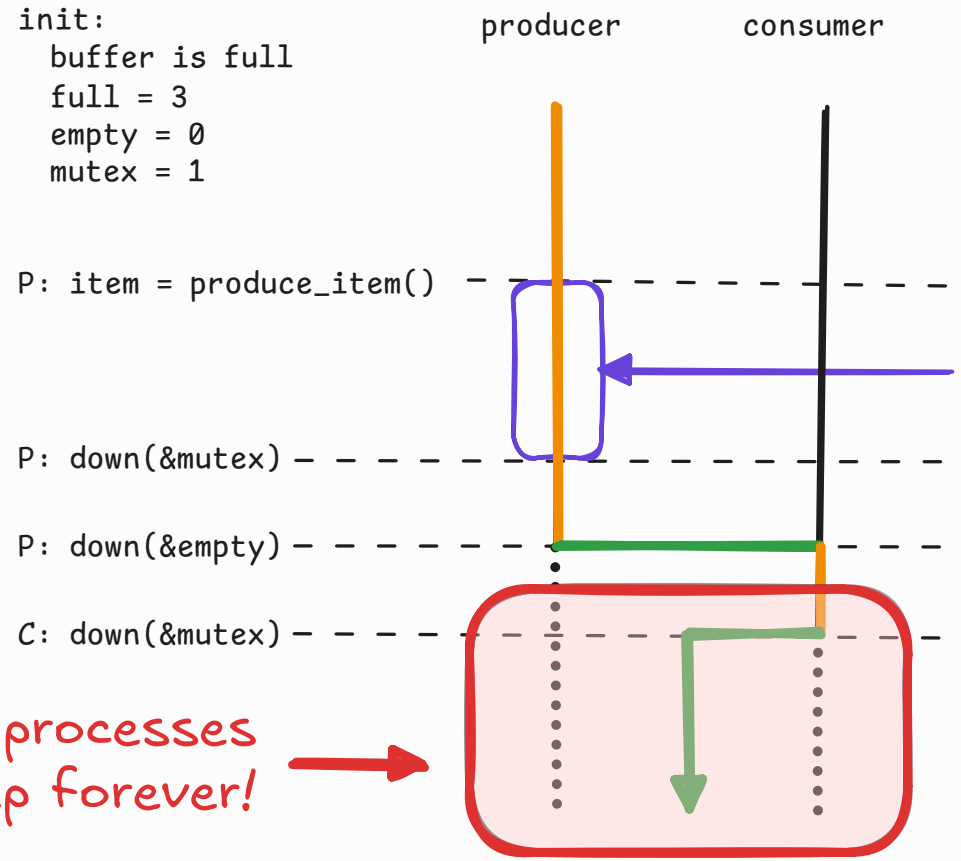
Does order matter?



```
1:  /** producer-consumer with 2 semaphores and a mutex */
2:
3:  #define N 3      // number of slots in buffer
4:  typedef int semaphore;
5:  semaphore full  = 0; // count full slots
6:  semaphore empty = N; // count empty slots
7:  semaphore mutex = 1; // lock buffer access
8:
9:  void producer(void) {
10:     int item;
11:     while (1) {
12:         item = produce_item();
13:         down(&mutex);
14:         down(&empty);
15:         insert_item(item);
16:         up(&full);
17:         up(&mutex);
18:     }
19: }
20:
21: void consumer(void) {
22:     int item;
23:     while (1) {
24:         down(&mutex);
25:         down(&full);
26:         item = remove_item();
27:         up(&empty);
28:         up(&mutex);
29:         consume_item(item);
30:     }
31: }
```

Would this work?

What would happen?



produce_item()

mutex = 0

(empty == 0) -> sleep()

(mutex == 0) -> sleep()

the producer waits forever for an empty slot, while the consumer waits for the mutex

both processes sleep forever!

Summary

- **Semaphores** are synchronization primitives used to count resources or events
  - They have 2 atomic operations: `up` and `down`
- **Binary Semaphores** are semaphores that can only be at 0 or 1
  - They act as mutual exclusion locks
- Three semaphores are required to resolve the producer-consumer problem
  - Counting full and empty slots, as well as mutual exclusion on buffer access
- The **order** of semaphore operations matters for preventing all processes from blocking