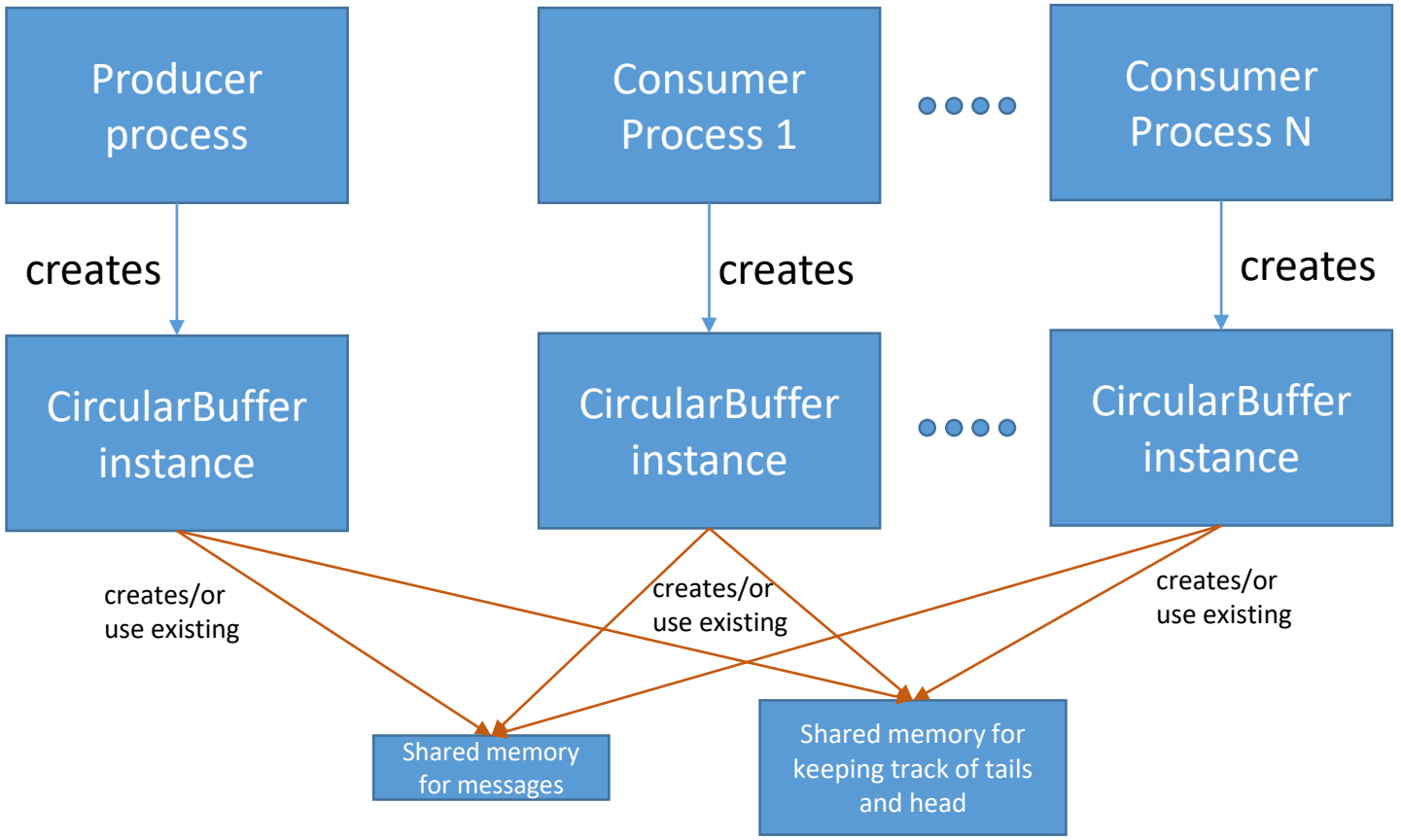
Circular buffer/queue description for UD1414 assignment

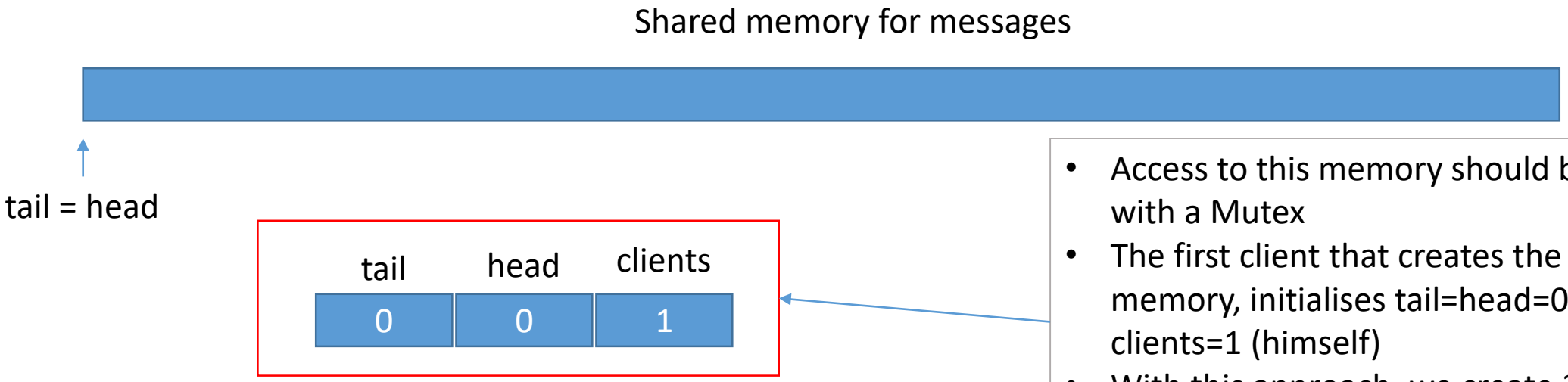
High level view



- One producer, several consumers start running.
- Each of them, creates an instance of a CircularBuffer object.
  - When the CircularBuffer is created, it checks whether the shared memory exists (already created by another CircularBuffer) or not.
  - If it does not exist, it creates it, otherwise simply “connects” to it.
- The first instance of CircularBuffer created, initialises the shared memory

Circular buffer details (alternative 1)

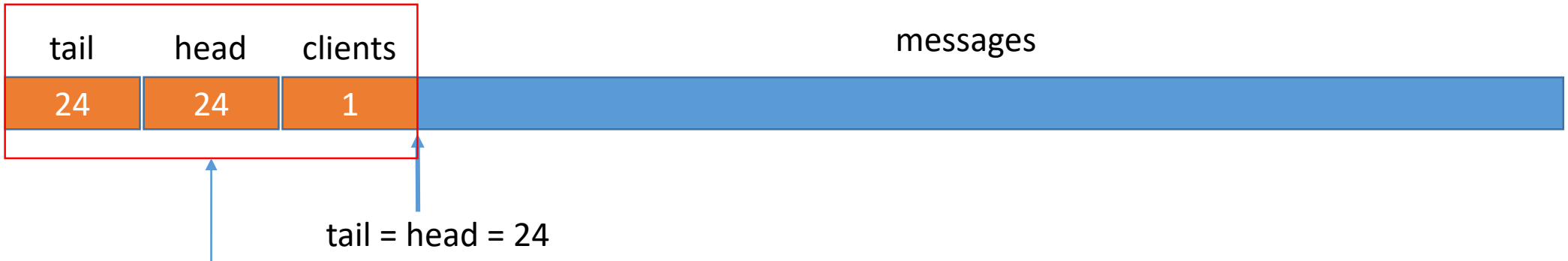
- Initial state, if using 2 separate blocks of shared memory.



- Access to this memory should be protected with a Mutex
- The first client that creates the shared memory, initialises tail=head=0 and clients=1 (himself)
- With this approach, we create 2 shared memory blocks, one for data, one for the “control” information (tail,head,etc).

Circular buffer details (alternative 2)

- Initial state, if using ONLY 1 block of shared memory.



- Access to this memory should be protected with a Mutex
- The first client that creates the shared memory, initialises tail=head=3*sizeof(size_t)=24 and clients=1 (himself)

If we use a single block of memory, then the memory address given by the OS when creating the block is used to keep the head, tail and other data and then after that comes the messages.
The available area for messages is the total area minus the tail,head and other Variables (24 bytes in this example) that we want to keep track of.

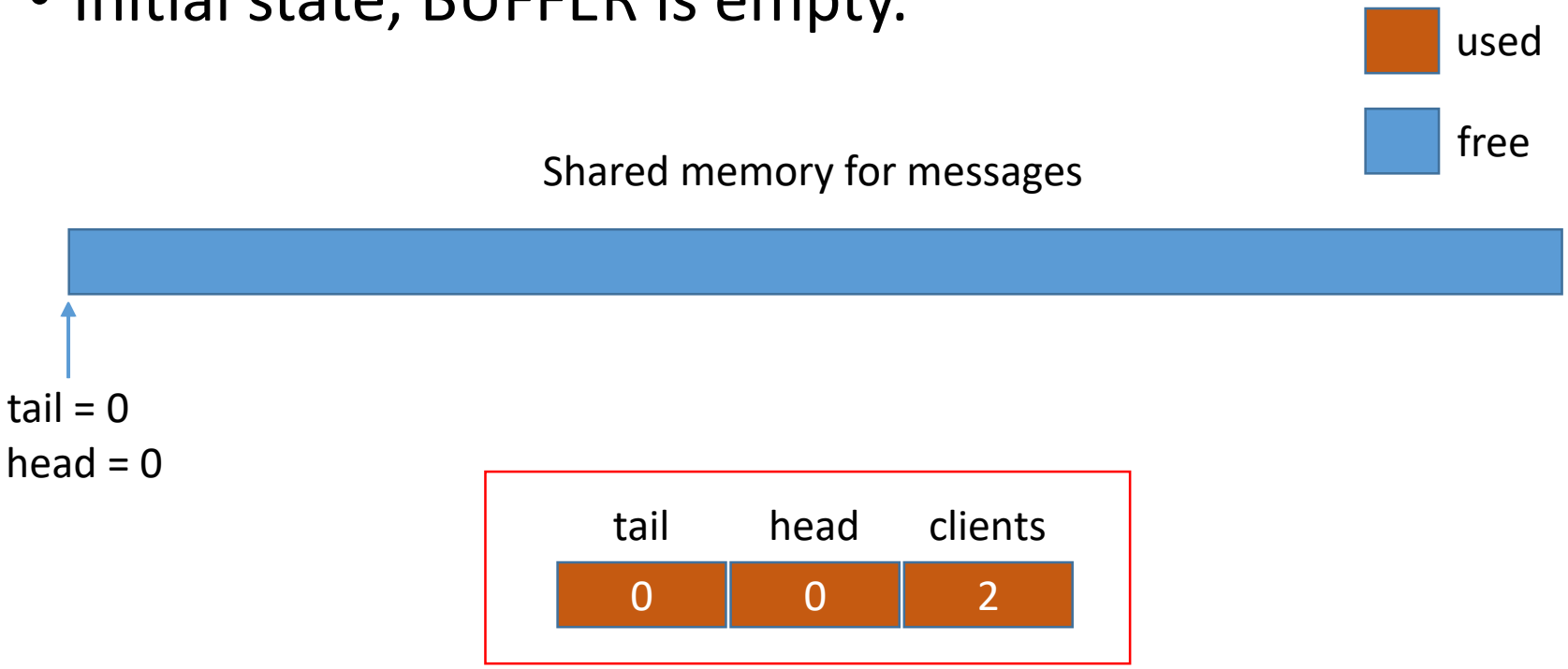
Circular buffer details

Disclaimer

- All examples from now on are going to be given using the first approach, 2 blocks, one for messages one for control data.
- In these examples, I assume that “head” is the location where the producer will write the next message, whilst “tail” points to the location where the consumer will read the next message.
- I will also assume that there are 1 producer and 2 consumers
- This is ONE possible approach, not the only one for implementing a circular buffer for IPC.

Circular buffer details

- Initial state, BUFFER is empty.

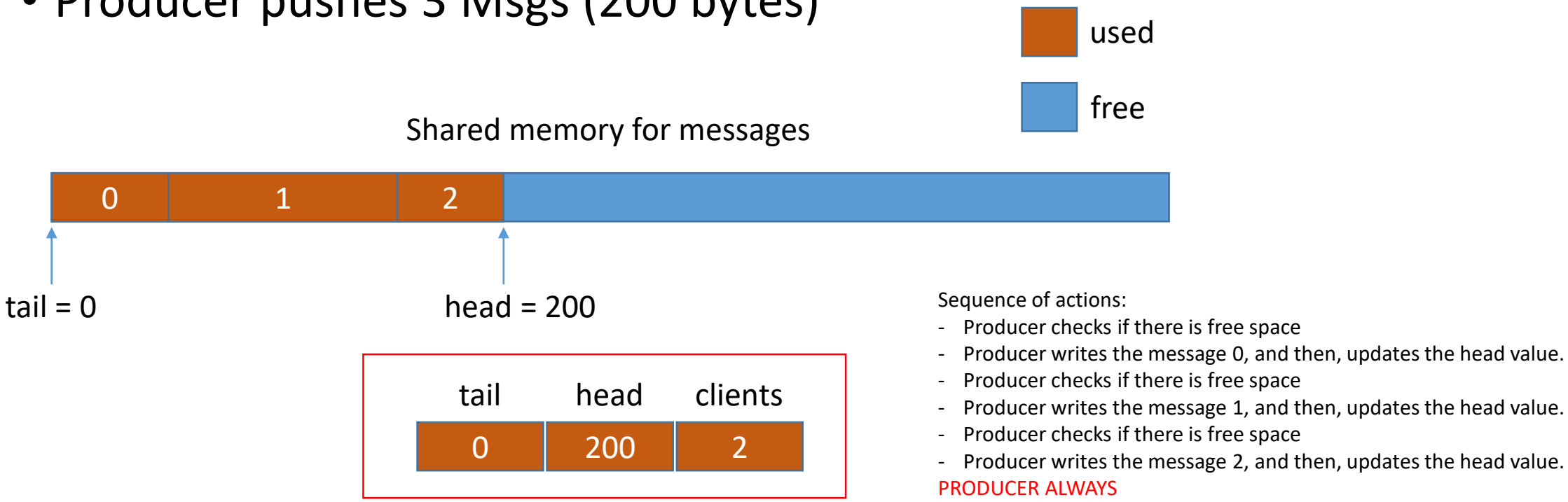


- This is the content of the shared memory, that everyone sees.

  2 consumers, 1 producer. No data.

Circular buffer details

- Producer pushes 3 Msgs (200 bytes)

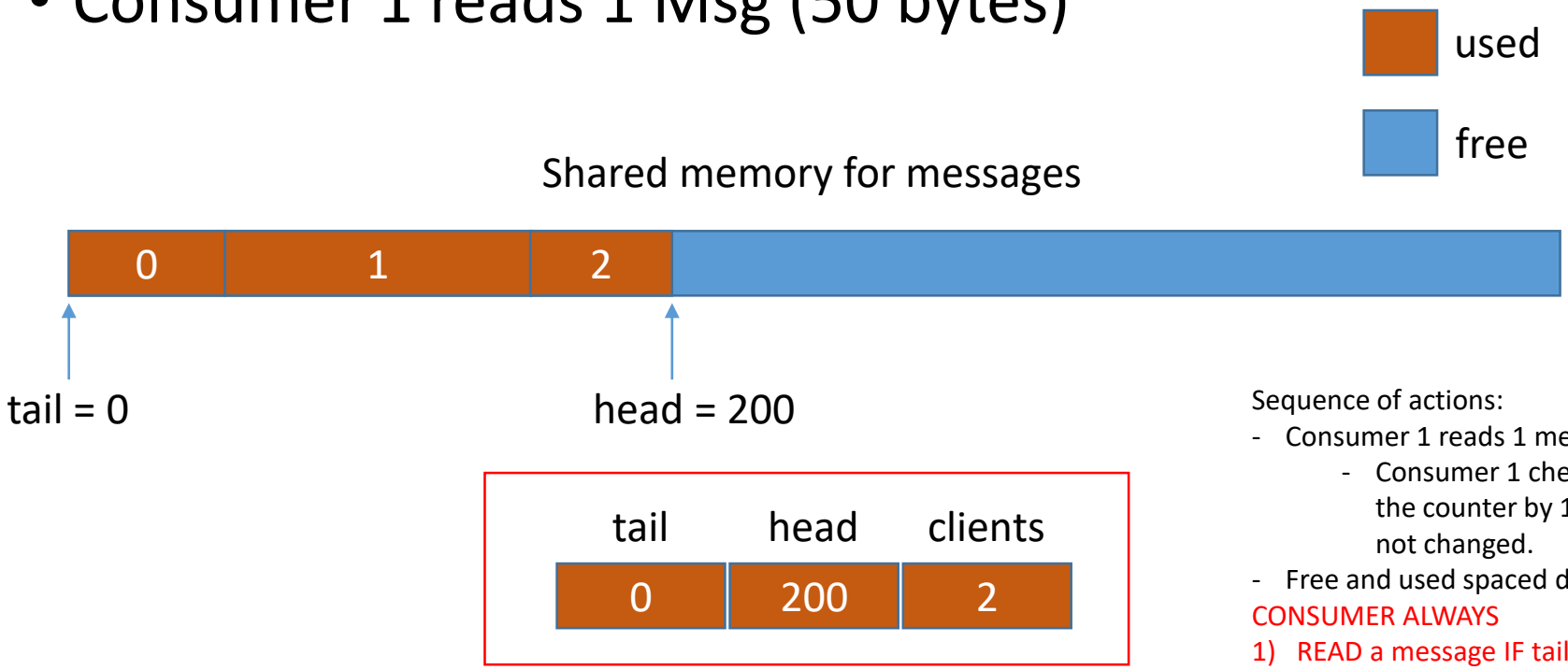


Sequence of actions:
- Producer checks if there is free space
- Producer writes the message 0, and then, updates the head value.
- Producer checks if there is free space
- Producer writes the message 1, and then, updates the head value.
- Producer checks if there is free space
- Producer writes the message 2, and then, updates the head value.

PRODUCER ALWAYS
1) CHECK FOR FREE SPACE,
2) WRITE THE MESSAGE.
3) UPDATE THE HEAD AFTER WRITING THE MESSAGE.

Circular buffer details

- Consumer 1 reads 1 Msg (50 bytes)



Sequence of actions:

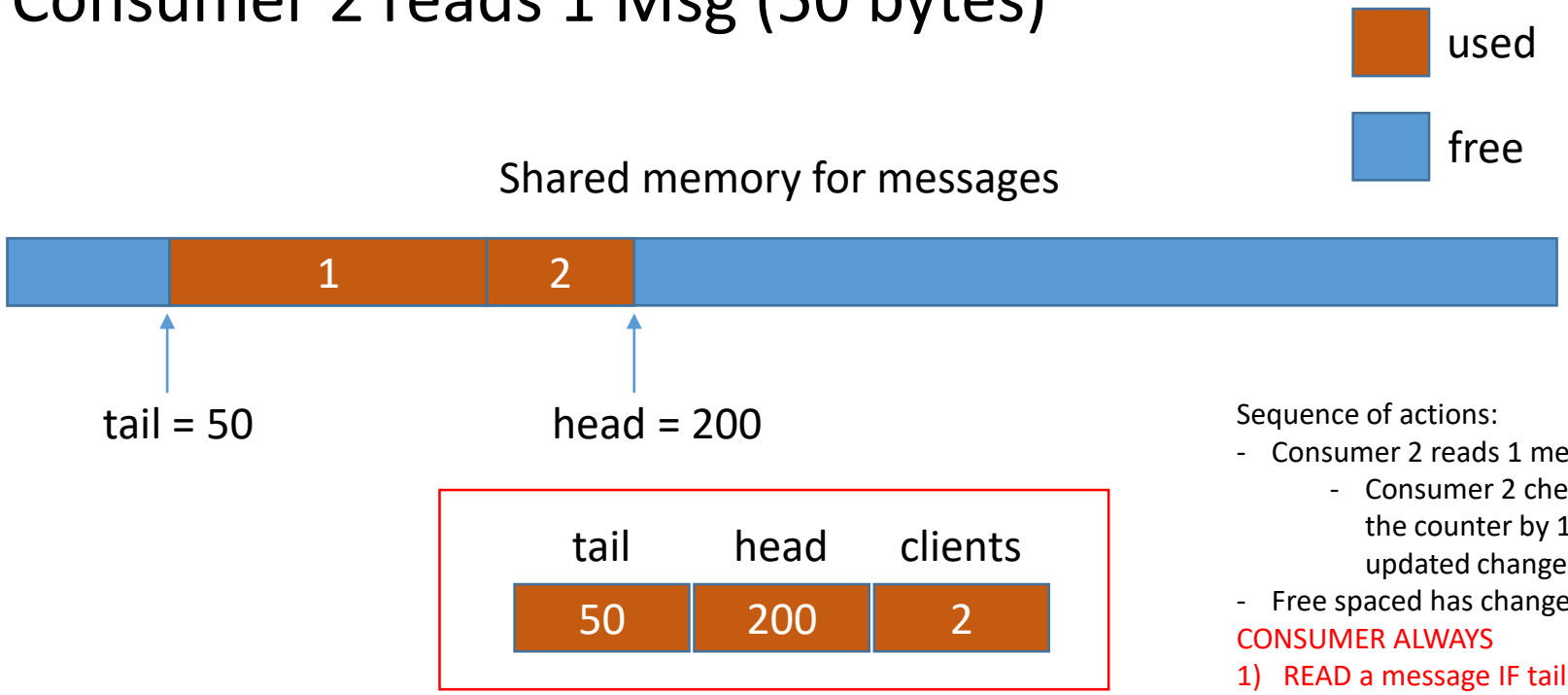
- Consumer 1 reads 1 message, moves its INTERNAL tail 50 bytes.
    - Consumer 1 checks the message counter (2) and decrements the counter by 1. The counter is not 0, so the shared tail is not changed.
- Free and used spaced did not change!

CONSUMER ALWAYS

1) READ a message IF tail!=head,
2) Decrement (using a LOCK) message clients counter.
3) If clients counter == 0, moves the tail to next message.

Circular buffer details

- Consumer 2 reads 1 Msg (50 bytes)



Sequence of actions:

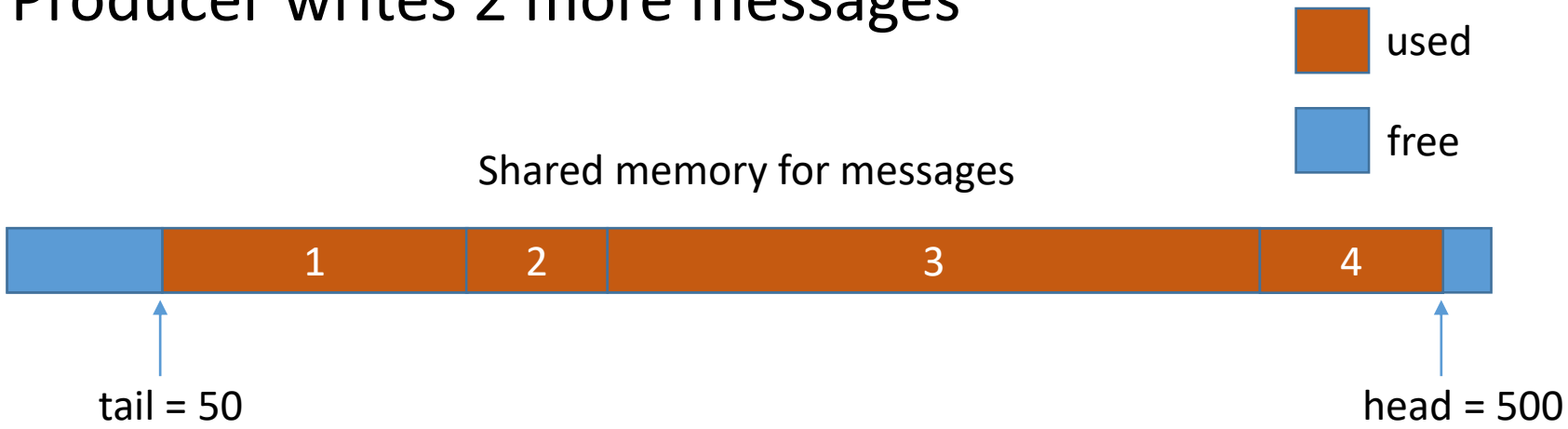
- Consumer 2 reads 1 message, moves its INTERNAL tail 50 bytes.
    - Consumer 2 checks the message counter (1) and decrements the counter by 1. The counter IS 0, so the shared tail is updated changed.
- Free spaced has changed, incrementing by 50 bytes!

CONSUMER ALWAYS

1) READ a message IF tail!=head,
2) Decrement (using a LOCK) message clients counter.
3) If clients counter == 0, moves the shared tail (using a LOCK) to next message.

Circular buffer details

- Producer writes 2 more messages



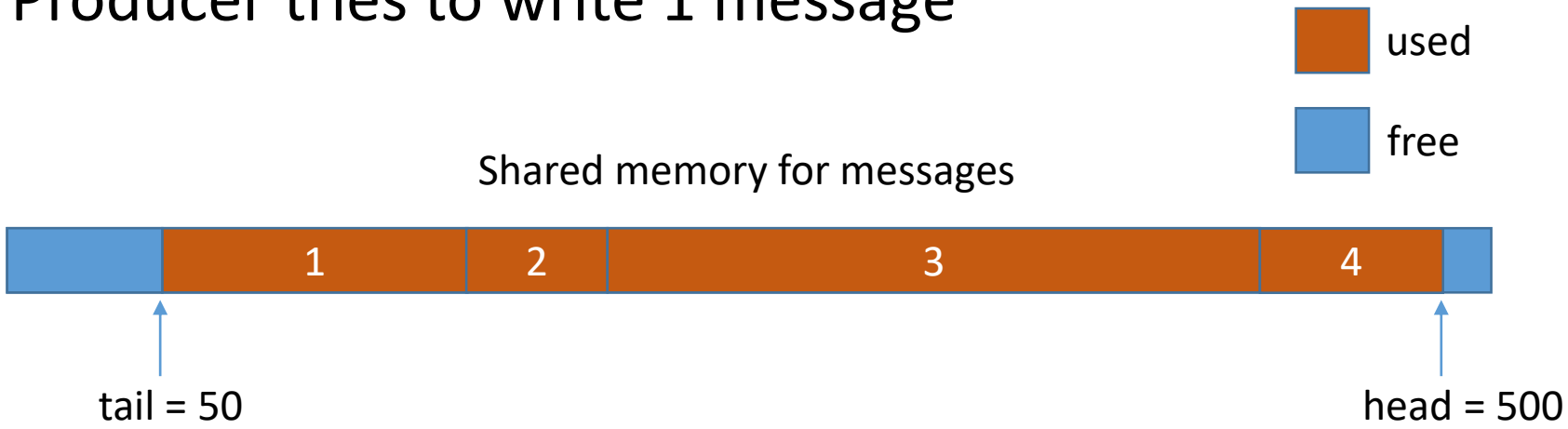
tail	head	clients
50	500	2

Sequence of actions:
- Producer checks if there is free space
- Producer writes the message 3, and then, updates the head value.
- Producer checks if there is free space
- Producer writes the message 4, and then, updates the head value.

PRODUCER ALWAYS
1) CHECK FOR FREE SPACE,
2) WRITE THE MESSAGE.
3) UPDATE THE HEAD AFTER WRITING THE MESSAGE.

Circular buffer details

- Producer tries to write 1 message



tail	head	clients
50	500	2

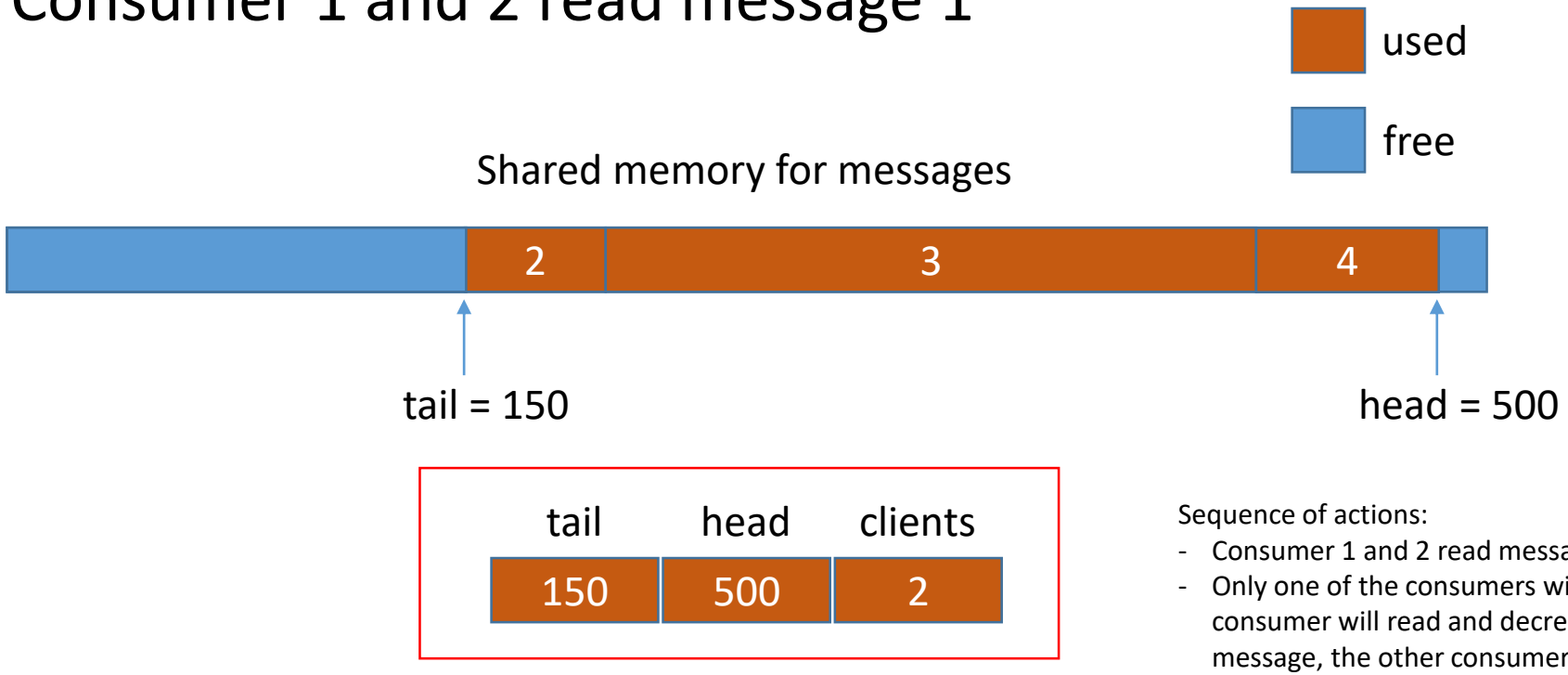
Sequence of actions:
- Producer checks if there is free space
  - The total space available between the head and the tail is not enough to write the new message.
- Producer "waits" and re-tries later.

Free space is calculated in the following way:

```
If (head >= tail)
   freespace = (TOTAL – head) + tail;
else
   freespace = tail – head;
```

Circular buffer details

- Consumer 1 and 2 read message 1

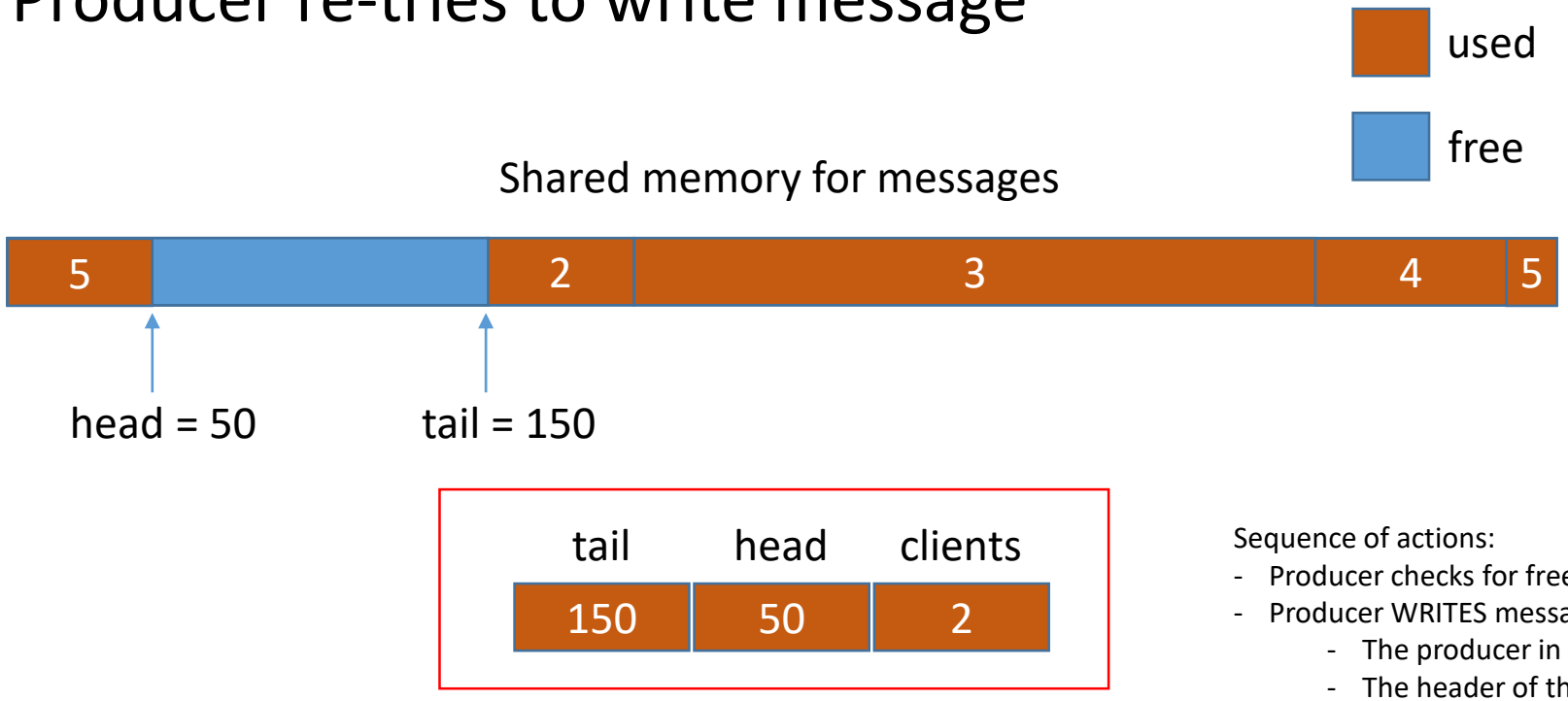


Sequence of actions:

- Consumer 1 and 2 read message 1
- Only one of the consumers will be able to acquire the LOCK first, this consumer will read and decrement the clients counter in the message, the other consumer comes afterwards and also LOCKS (after the LOCK has been released) and decrements, reaching 0.
- The consumer that came 2nd, moves the shared tail freeing up space.

Circular buffer details

- Producer re-tries to write message



Sequence of actions:
- Producer checks for free space, and now there is enough space.
- Producer WRITES message in the free space
  - The producer in this case SPLITS the message in two parts.
  - The header of the message fits in the first half ALWAYS, so in the first part there is enough information for consumers to determine the real size of the message.
- Producer updates shared “head”.

Now, as it can be seen clearly in the drawing, the free space is simply:
freespace = tail – head;

Circular buffer details

- Other details
  - If you have noticed, I have mentioned
    - “INTERNAL” tail
    - “Shared” tail
  - I have also mentioned a “clients counter” in the message header.
    - The “clients counter” is part of the message header, and is always initialised for every new message, to the number of consumers connected to the system.
    - The “INTERNAL” tail is a local variable (not shared memory) that a consumer uses to track where is the next message to be read.
    - The “Shared” tail, stored in shared memory, indicates where is the last message to be read by at least one consumer. It is used by the producer to check for free space.
    - The “Shared” tail is only moved when a consumer reads a message (after LOCKING) and identifies that the “clients counter” is 1.
      - After the shared tail is moved, space is freed and the producer can use that for new messages.