Synchronisation problems

Module 4 self study material

Bounded buffer

Reader and writers

Priority inversion

Operating systems 2020

1DT003, 1DT044 and 1DT096

karl.marklund@it.uu.se

Uppsala university

Classical problems of synchronization

- The bounded buffer problem
- The readers and writers problem
- Priority inversion

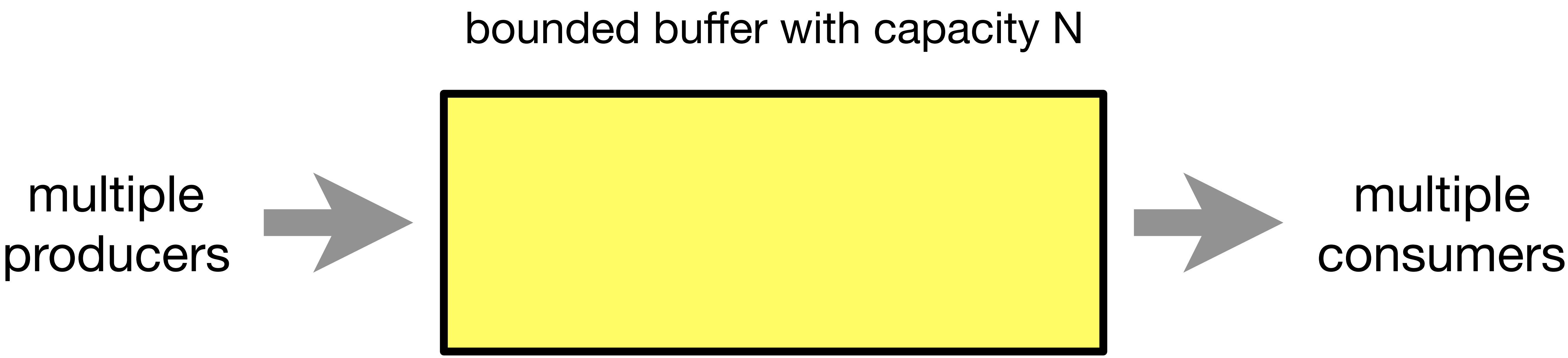
Bounded buffer

Bounded buffer

A bounded buffer lets multiple **producers** and multiple **consumers** share a single buffer. Producers write data to the buffer and consumers read data from the buffer.

★ **Producers** must **block** if the buffer is **full**.

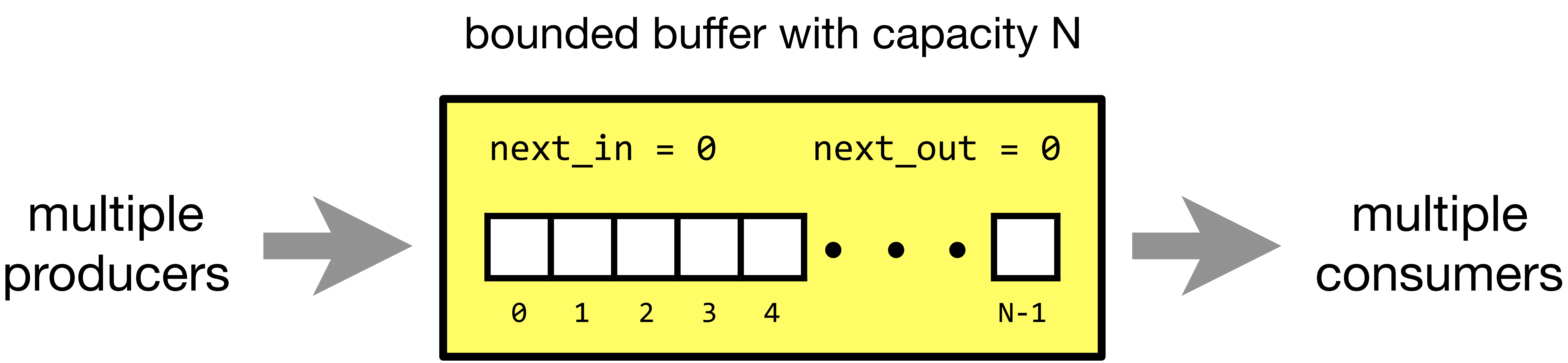
★ **Consumers** must **block** if the buffer is **empty**.



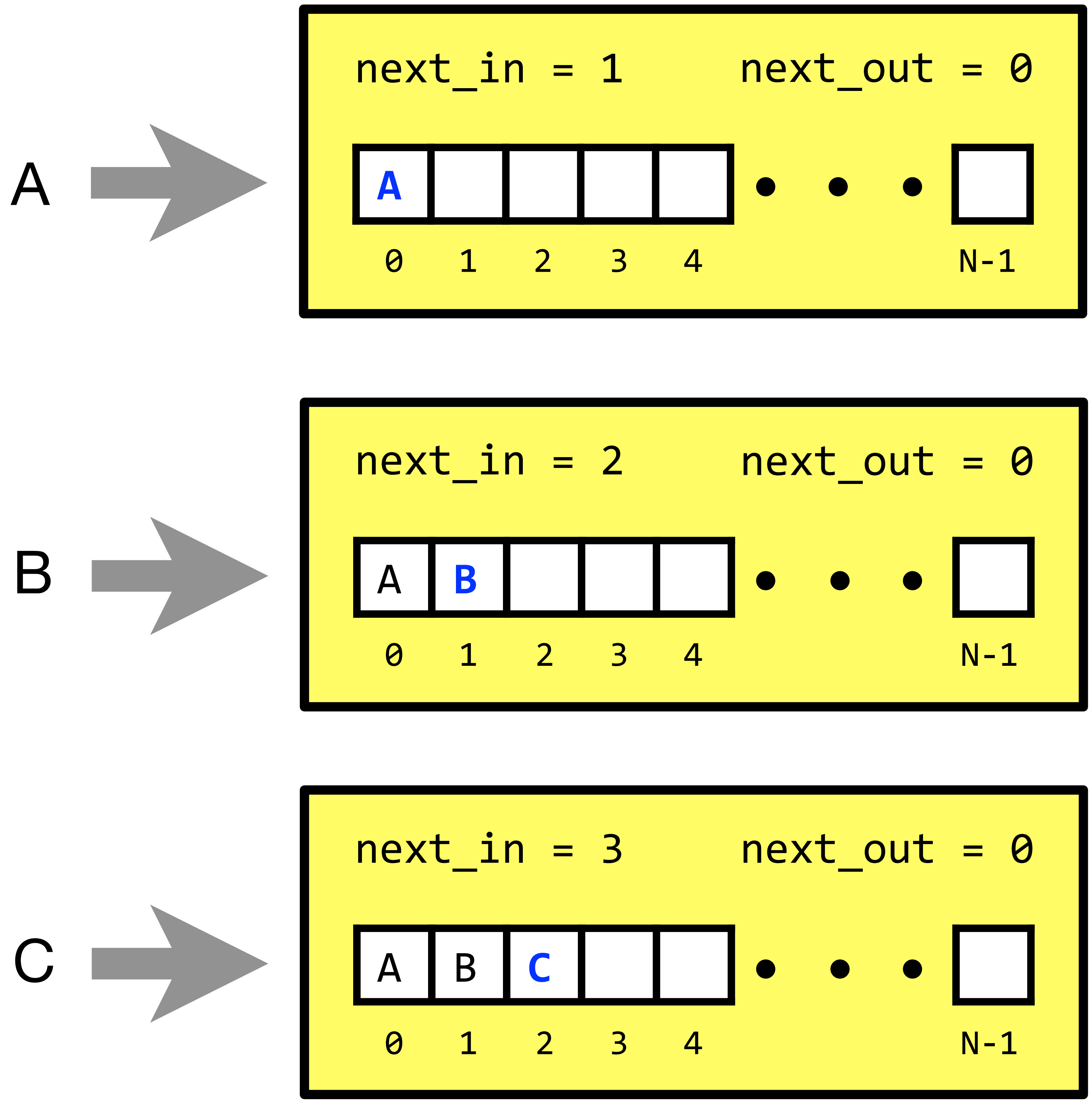
Implementation

Use an **array** of size **N** to store the data items in the buffer.

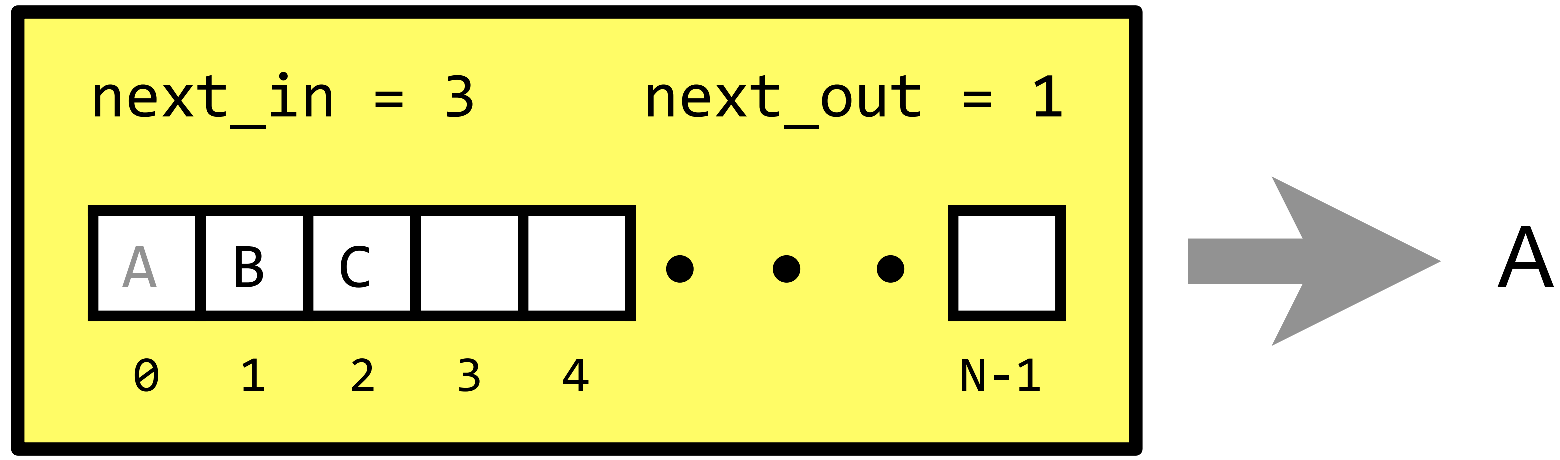
- Keep track of where to produce the next data item using index `next_in`.
- Keep track of from where to consume the next data item using index `next_out`.

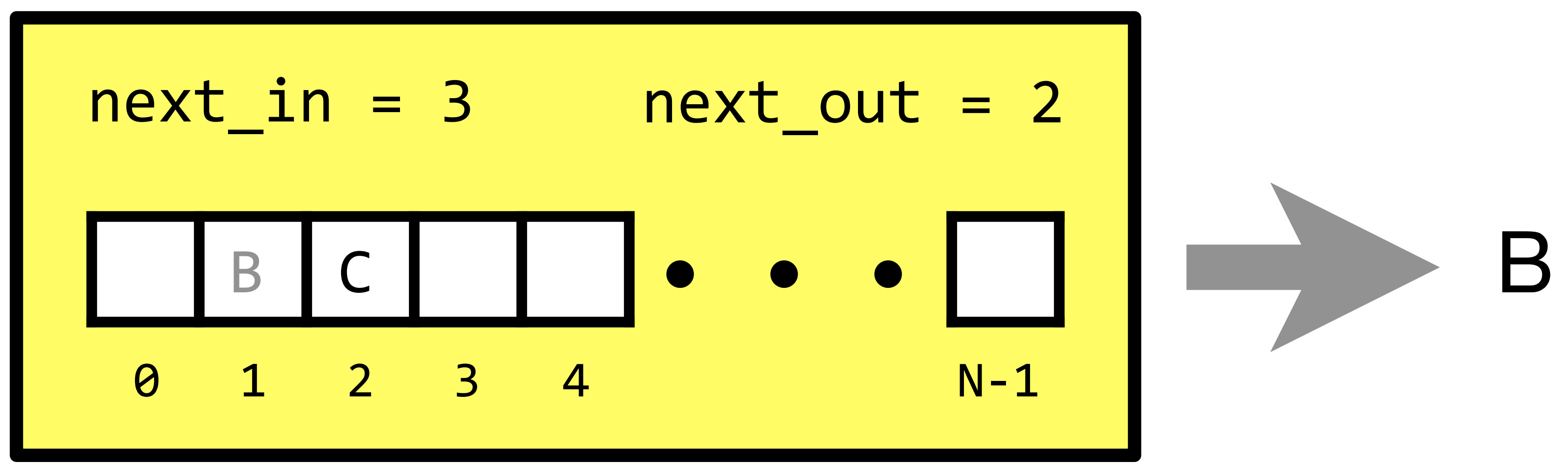


The **next_in** index must be **incremented** after every **write** to the buffer.

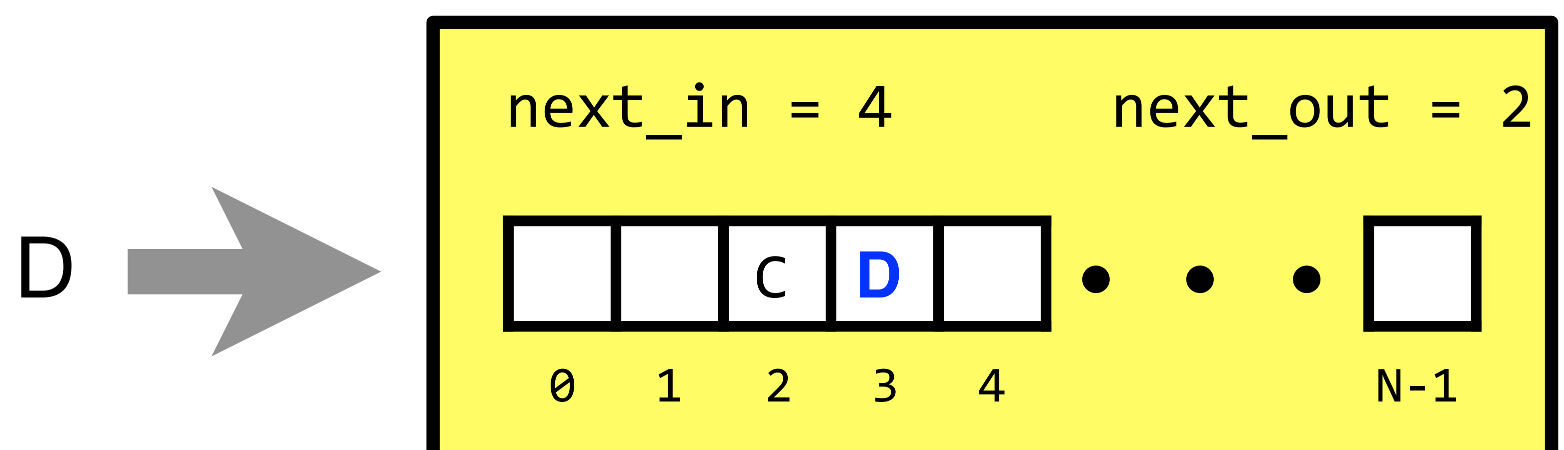


The **next_out** index must be **incremented** after every **read** from the buffer.

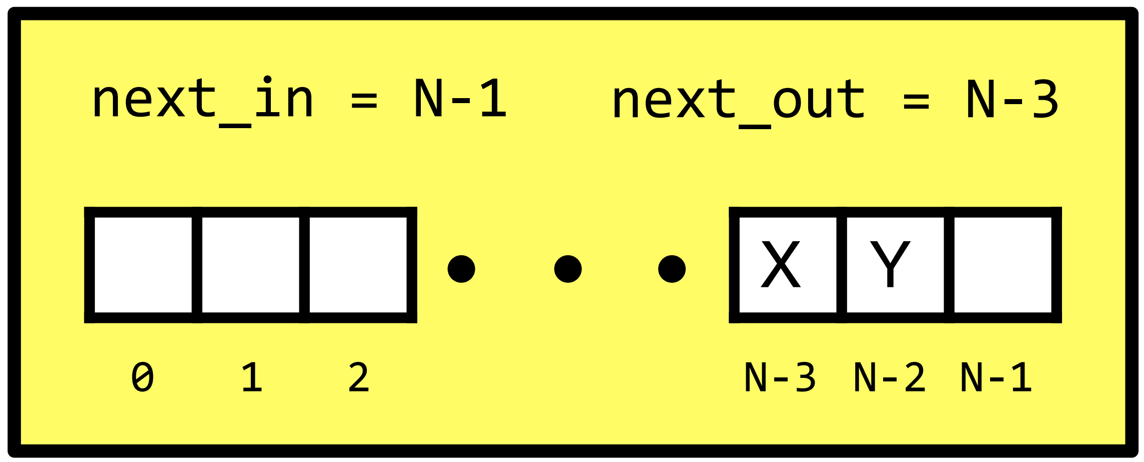




Let's make an additional write to the buffer.

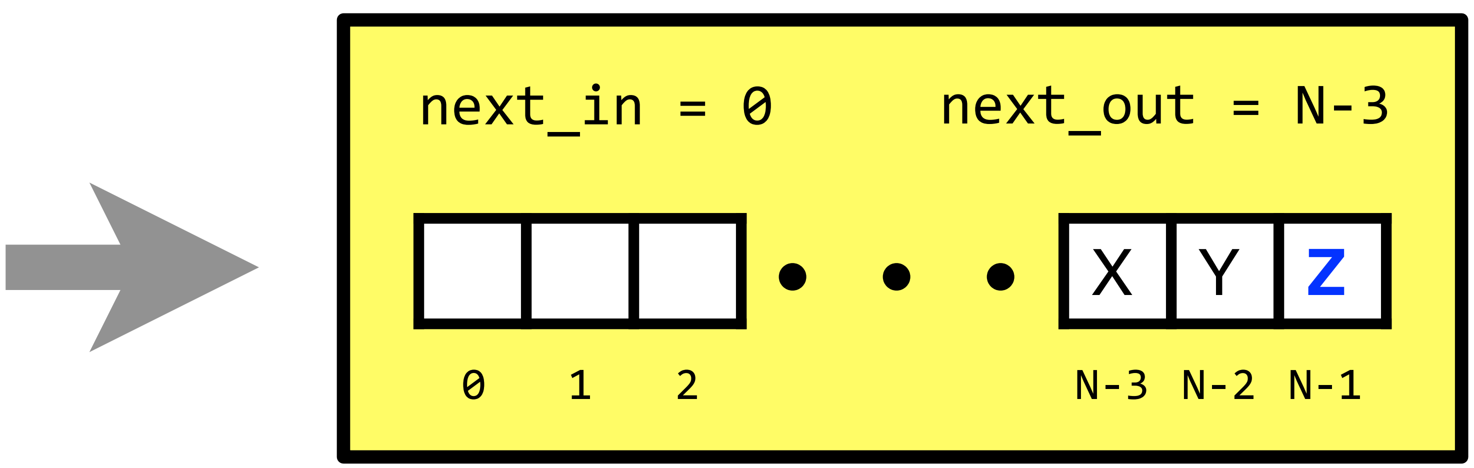


The buffer **wraps around** in a **circular** manner.



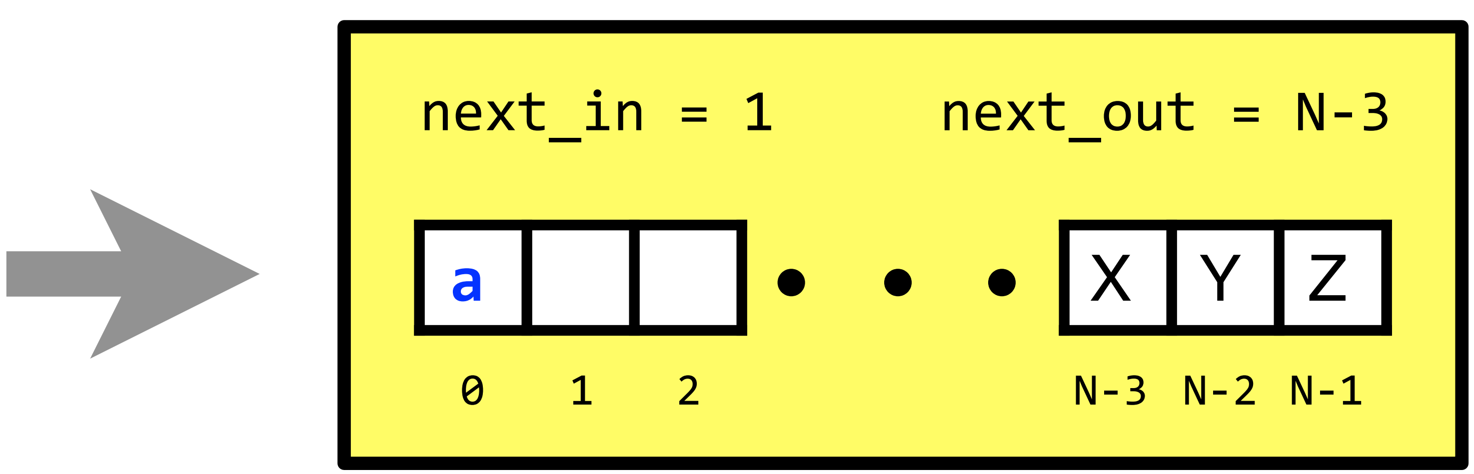
Assume the buffer is in the following state. The next write will be to the last element of the array.

Z



The next write will be to the first element of the array.

a



The next write will be to the second element of the array.

Wrap around

Use the modulo operator `%` to make the index `next_in` wrap around after `N` writes and the index `next_out` wrap around after `N` reads.

Producer

```
next_in = (next_in + 1) % N
```

Consumer

```
next_out = (next_out + 1) % N
```

Mutual exclusion

All updates to the buffer state must be done in a critical section. More specifically, mutual exclusion must be enforced between the following **critical sections**:

- A **producer writing** to a buffer slot and **updating** `next_in`.
- A **consumer reading** from a buffer slot and **updating** `next_out`

A **binary semaphore** or a **mutex lock** can be used to protect access to the critical sections.

Synchronisation

Producers must **block** if the buffer is **full**. **Consumers** must **block** if the buffer is **empty**.

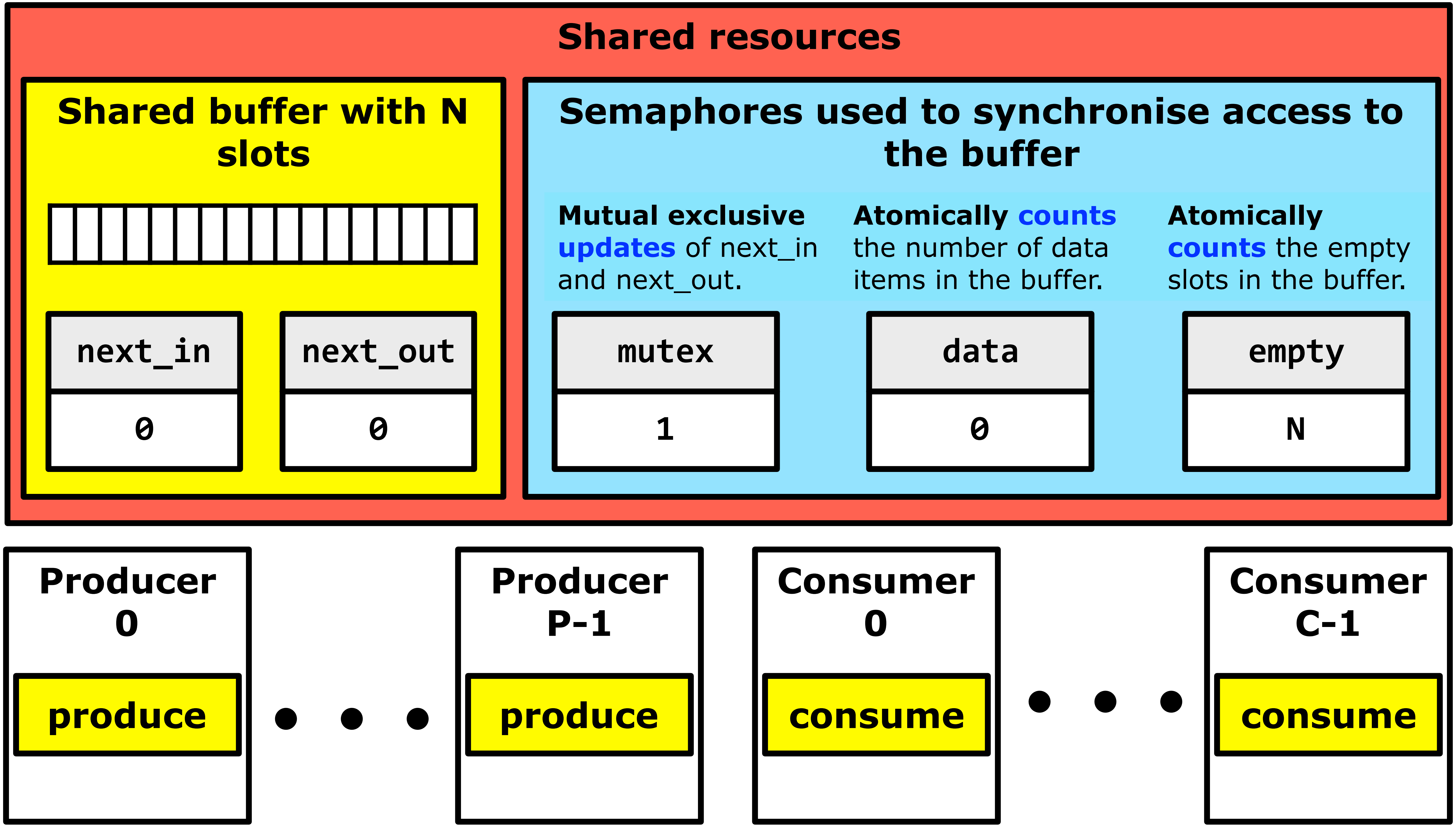
Use one **semaphore** named `empty` to count the empty slots in the buffer.

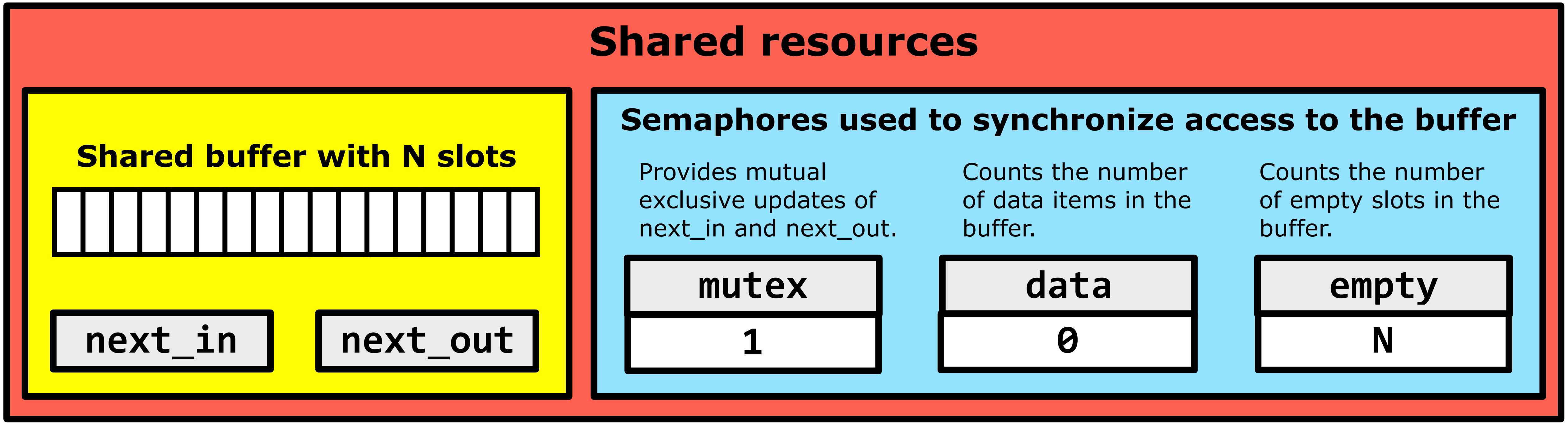
- **Initialise** this semaphore to `N`.
- A **producer** must **wait** on this semaphore before writing to the buffer.
- A **consumer** will **signal** this semaphore after reading from the buffer.

Use one **semaphore** named `data` to count the number of data items in the buffer.

- **Initialise** this semaphore to `0`.
- A **consumer** must **wait** on this semaphore before reading from the buffer.
- A **producer** will **signal** this semaphore after writing to the buffer.

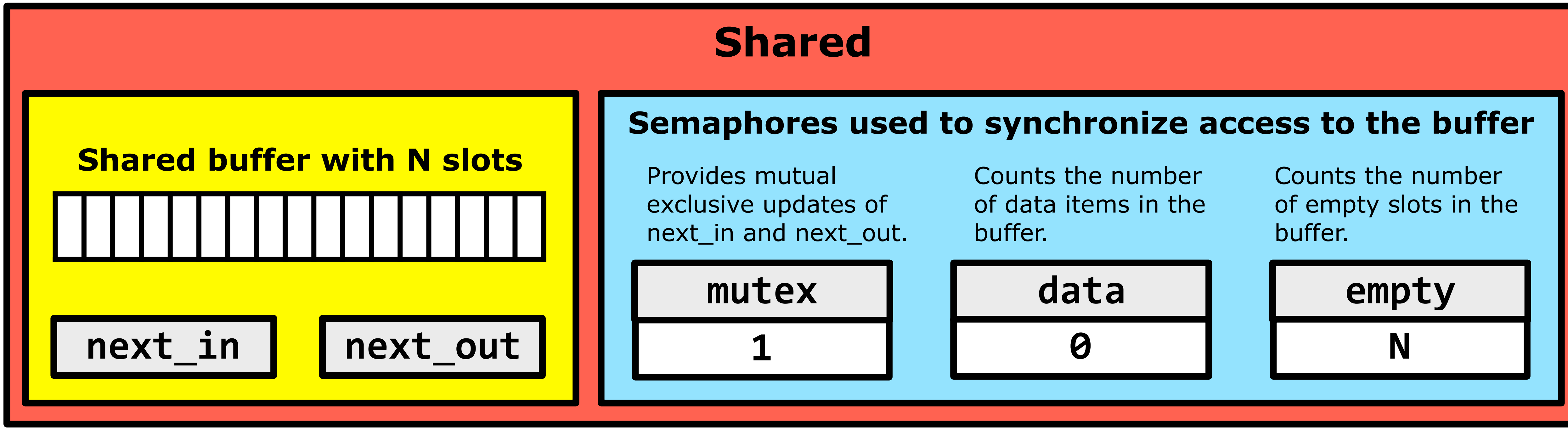
P producers and **C** consumers using a shared bounded buffer of size **N**. Producers writes to buffer at index `next_in` and consumer reads at index `next_out`.





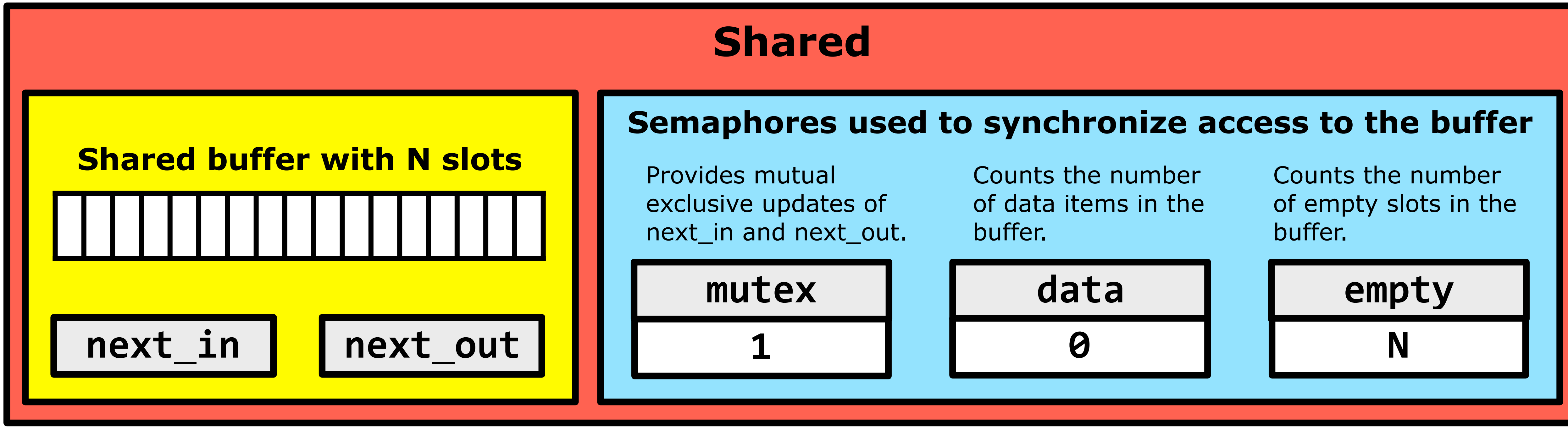

```
produce(buffer, *data) {
1  wait(empty)
2  wait(mutex)
3  buffer[next_in] = copy(data)
4  next_in = nexp_in + 1 % N
5  signal(mutex)
6  signal(data)
}
```

1. **Block** if buffer is **full**, otherwise atomically decrement the empty counter.
2. **Enter critical section**, i.e., make sure no other producer or consumer updates the buffer at the same time.
3. **Copy** data **to** slot in **buffer**.
4. **Update next_in**.
5. **Leave** the **critical section**.
6. **Atomically increment** the **data counter**.




```
consume(buffer, *data) {
  wait(data)
  wait(mutex)
  data = copy(buffer[next_out])
  next_out = next_out + 1 % N
  signal(mutex)
  signal(empty)
}
```

1. **Block** if buffer is **empty**, otherwise atomically decrement data counter.
2. **Enter critical section**, i.e., make sure no other producer or consumer updates the buffer at the same time.
3. **Copy** data **from** slot in **buffer**.
4. **Update next_out**.
5. **Leave** the **critical section**.
6. **Atomically increment** the **empty counter**.




```
produce(buffer, *data) {
  wait(empty)
  wait(mutex)

  buffer[next_in] = copy(data)
  next_in = next_in + 1 % N

  signal(mutex)
  signal(data)
}
```

```
consume(buffer, *data) {
  wait(data)
  wait(mutex)

  data = copy(buffer[next_out])
  next_out = next_out + 1 % N

  signal(mutex)
  signal(empty)
}
```

A pipe is a bounded buffer

```
ls | grep .txt | wc
```

Concurrent writes to a pipe

Is a single write to a pipe atomic, i.e., is the whole amount written in a single write operation not interleaved with data written by any other process?

POSIX.1-200

- **Using `write()` to write less than `PIPE_BUF` bytes must be atomic:** the output data is written to the pipe as a contiguous sequence.
- Writes of **more than `PIPE_BUF` bytes may be nonatomic:** the kernel may interleave the data, on arbitrary boundaries, with data written by other processes.

The value if `PIPE_BUF` is defined by each implementation, but the minimum is `512` bytes (see `limits.h`).

On Linux:

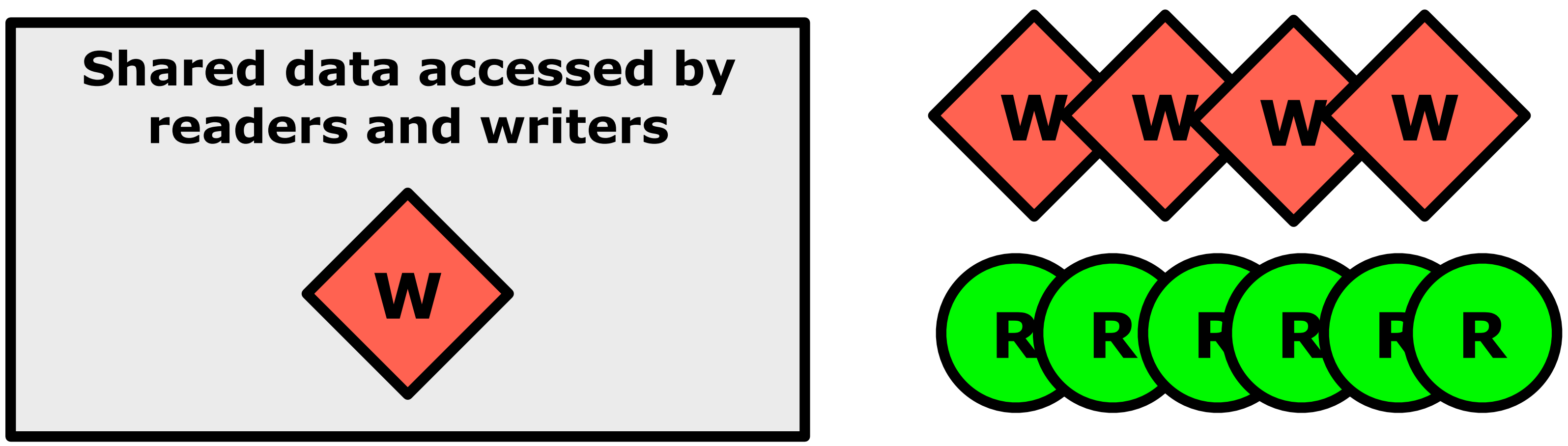
- `PIPE_BUF = 4096`.
- The value of `PIPE_BUF` is a consequence of other logic in the kernel, it is not a configuration parameter.

Readers and writers

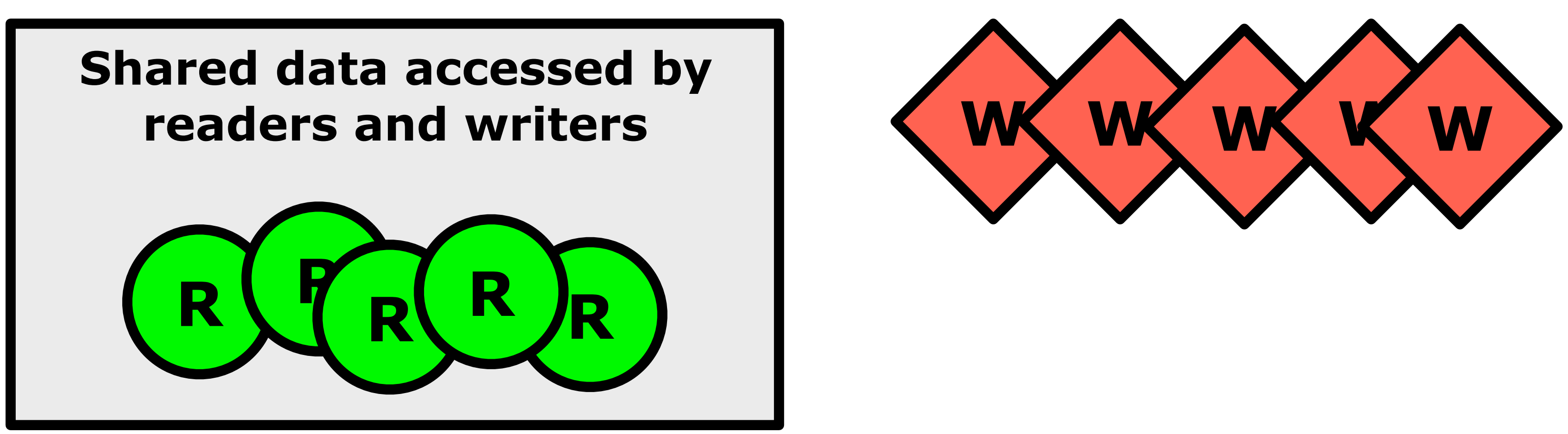
Readers-Writers Problem

A data set is shared among a number of concurrent processes. Readers only read the data set; they do not perform any updates. Writers can both read and write.

Only one single writer can access the shared data at the same time, any other writers or readers must be blocked.

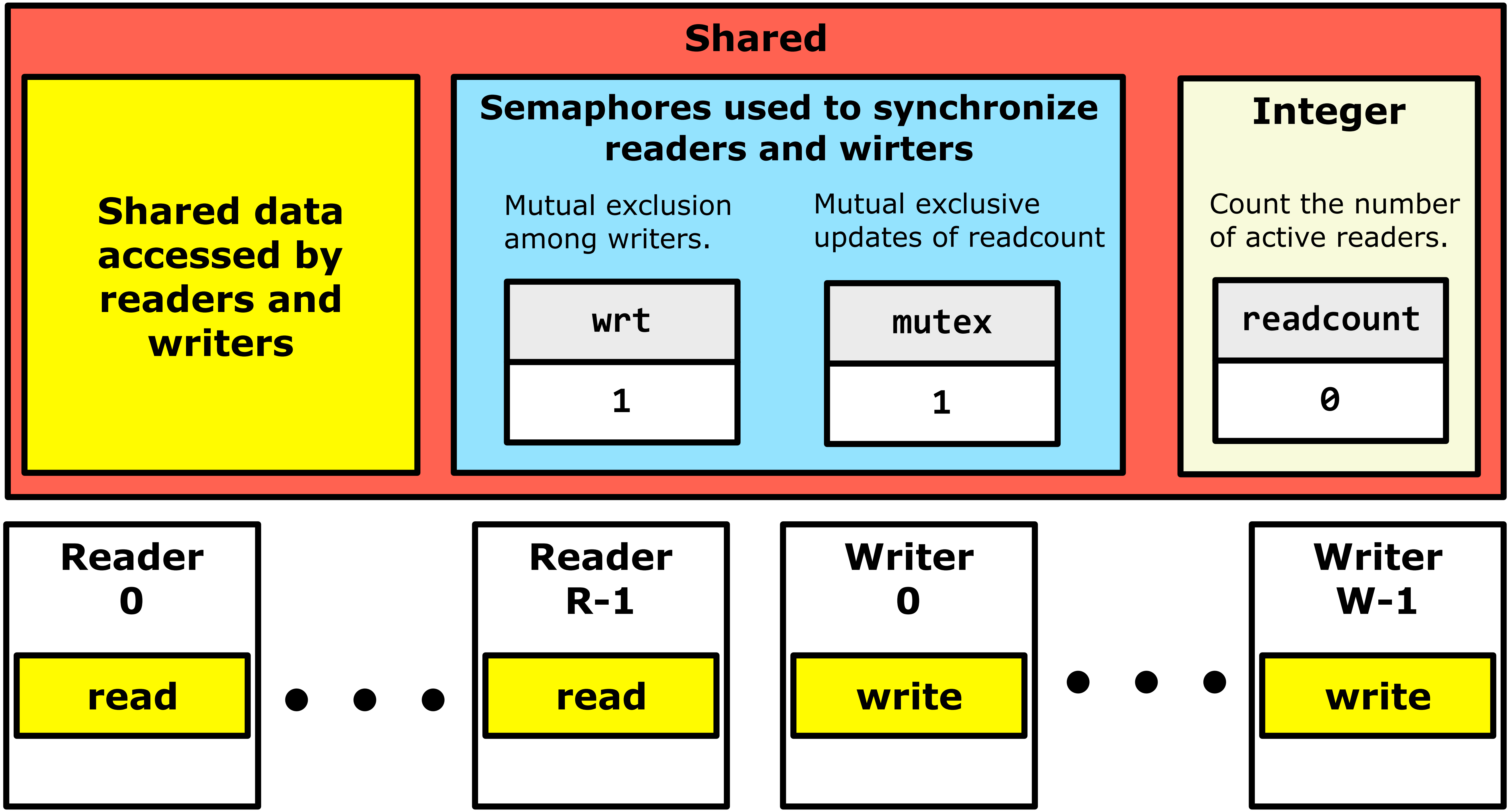


Allow multiple readers to read at the same time, any writers must be blocked



`R` readers and `W` writers access the same shared data set.

Allow multiple readers to read at the same time.
Only one single writer can access the shared data at the same time.



Shared

Shared data accessed by readers and writers

Semaphores used to synchronize readers and wirters

Mutual exclusion among writers.

`wrt`

Mutual exclusive updates of readcount

`mutex`

Integer

Count the number of active readers.

`readcount`

```
write(buffer, *data) {
 1  wait(wrt);

 2  // Write shared data

 3  signal(wrt);
}
```

1. Enter critical section, block if other task is writing.
2. Inside critical section, write to shared data structure.
3. Leave critical section.

```
read(buffer, *data) {
  wait(mutex);

  readcount++;
  if readcount == 1:
     wait(wrt);

  signal(mutex);

  // Read shared data

  wait(mutex);

  reacount--;
  if readcount == 0:
     signal(wrt);

  signal(mutex);
}
```

Semaphores

mutex   wrt

Integral counter

readcount

Entering

All readers need to mutually exclusively increment readcount when entering.

The first reader also need to block if a writer is active.

Leaving

All readers need to mutually exclusively decrement readcount when leaving.

The last reader also need to unblock any waiting writer.

Readers-Writers Problem

A data set is shared among a number of concurrent processes.

- Only one single writer can access the shared data at the same time, any other writers or readers must be blocked.
- Allow multiple readers to read at the same time, any writers must be blocked.

Semaphores mutex and **wrt**, both initialized to 1.

Integer readcount initialized to 0.

```
write(buffer, *data) {
  wait(wrt);

  // Write shared data

  signal(wrt);
}
```

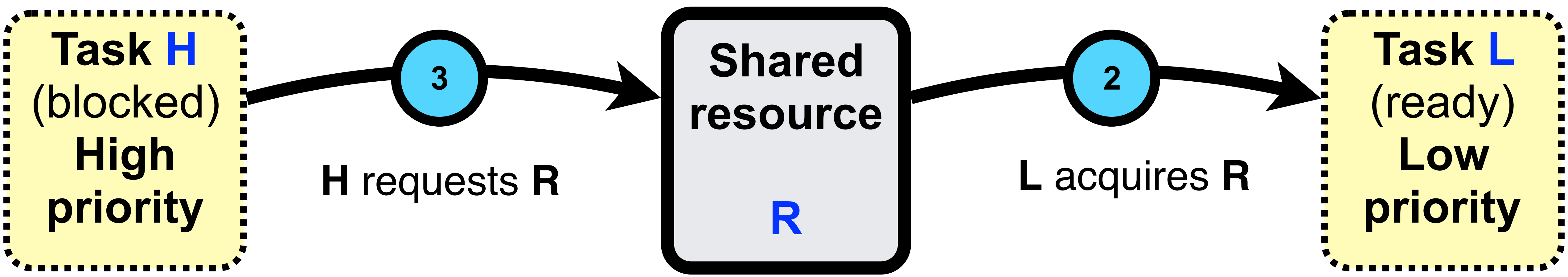
```
read(buffer, *data) {
  wait(mutex);

  readcount++;
  if readcount == 1:
     wait(wrt);

  signal(mutex);

  // Read shared data

  wait(mutex);

  readcount--;
  if readcount == 0:
     signal(wrt);

  signal(mutex);
}
```

Priority inversion

Scenario (1)

A **high priority task H** is blocked due to a **low priority task L** holding a **shared resource R** (for example a binary semaphore) task **H** wants to acquire.

1) Consider two tasks H and L, of high and low priority respectively, either of which can acquire exclusive use of a shared resource R.
2) L acquires R.
3) If H attempts to acquire R after L has acquired it, then H becomes blocked until L relinquishes the resource.



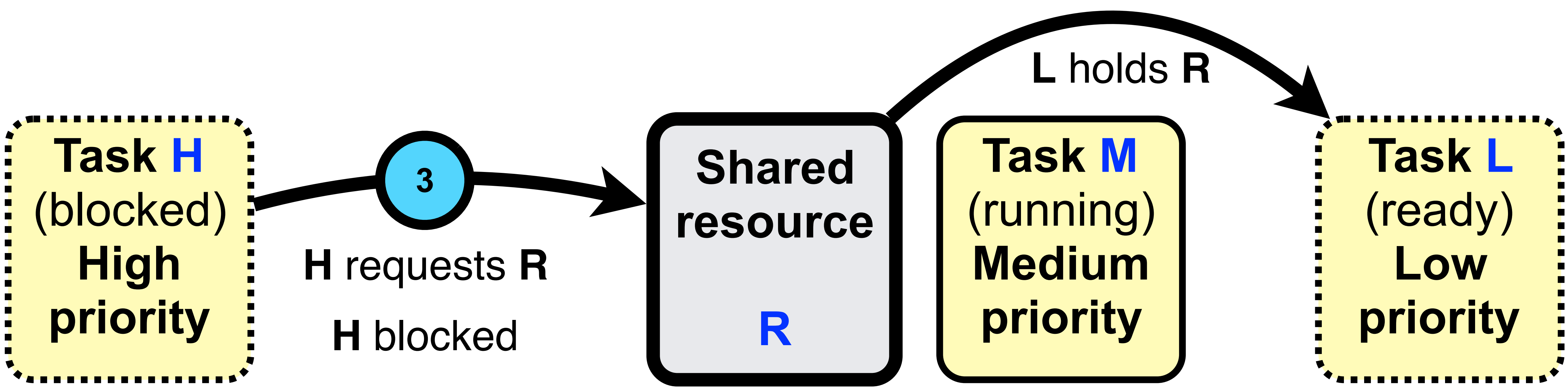
4) Sharing an exclusive-use resource (R in this case) in a well-designed system typically involves L relinquishing R promptly so that H (a higher priority task) does not stay blocked for excessive periods of time.

Scenario (2)

Let's introduce a third task **M** with **medium priority**, i.e., a priority between **high priority task H** and **low priority task L**.

Task M becomes ready (to run) during L's use of R.

1) M being higher in priority than L preempts R, causing L to not be able to relinquish R promptly.
2) H becomes ready to run.
3) H request to acquire R.
4) H (the highest priority process) becomes blocked since H cannot acquire R hold by L but L is not running (preempted by M).

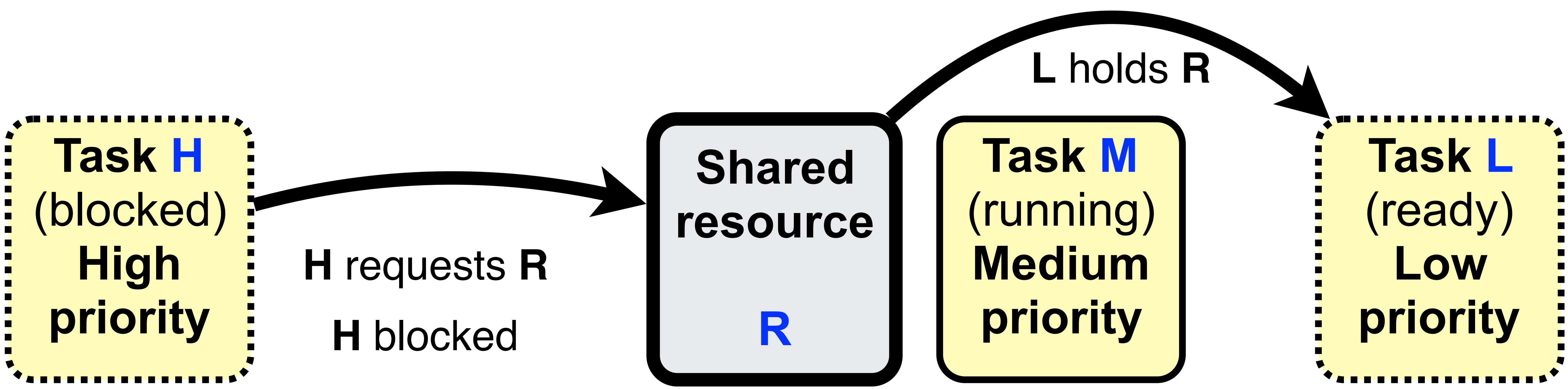


Priority inversion

A higher priority task is “preempted” by a lower priority one.

A **medium priority task** **M** preempts a **low priority task** **L** holding a **shared resource** **R**. A **high priority task** **H** is not able to run, although it has higher priority than **M** and **H** and **M** does not compete for **R**.

Solution to the priority inversion problem?



Priority inheritance protocol

When a task blocks one or more high-priority tasks, it ignores its original priority assignment and executes its critical section at an elevated priority level. After executing its critical section and releasing its locks, the process returns to its original priority level.

- ★ Suppose **H** is blocked by **L** for some shared resource **R**.
- ★ The priority inheritance protocol requires that **L** executes its critical section at **H**'s (high) priority.
- ★ As a result, **M** will be unable to preempt **L** and **M** will be blocked.
- ★ That is, the higher-priority job **M** must wait for the critical section of the lower priority job **L** to be executed, because **L** has inherited **H**'s priority.
- ★ When **L** exits its critical section, it regains its original (low) priority and awakens **H** (which was blocked by **L**).
- ★ **H**, having high priority, preempts **L** and runs to completion. This enables **M** and **L** to resume in succession and run to completion.

Priority inheritance and mutexes

What if a higher priority task is blocked on a mutex hold (owned) by a lower priority task?

- By default, if a task with a higher priority than the mutex owner attempts to lock a mutex, then the effective priority of the current owner is increased to that of the higher-priority blocked thread waiting for the mutex.
- The current owner's effective priority is again adjusted when it unlocks the mutex; its new priority is the maximum of its own priority and the priorities of those threads it still blocks, either directly or indirectly.