Synchronisation problems

Module 4 self study material

Bounded buffer

Reader and writers

Priority inversion

Operating systems 2018

1DT044 and 1DT096

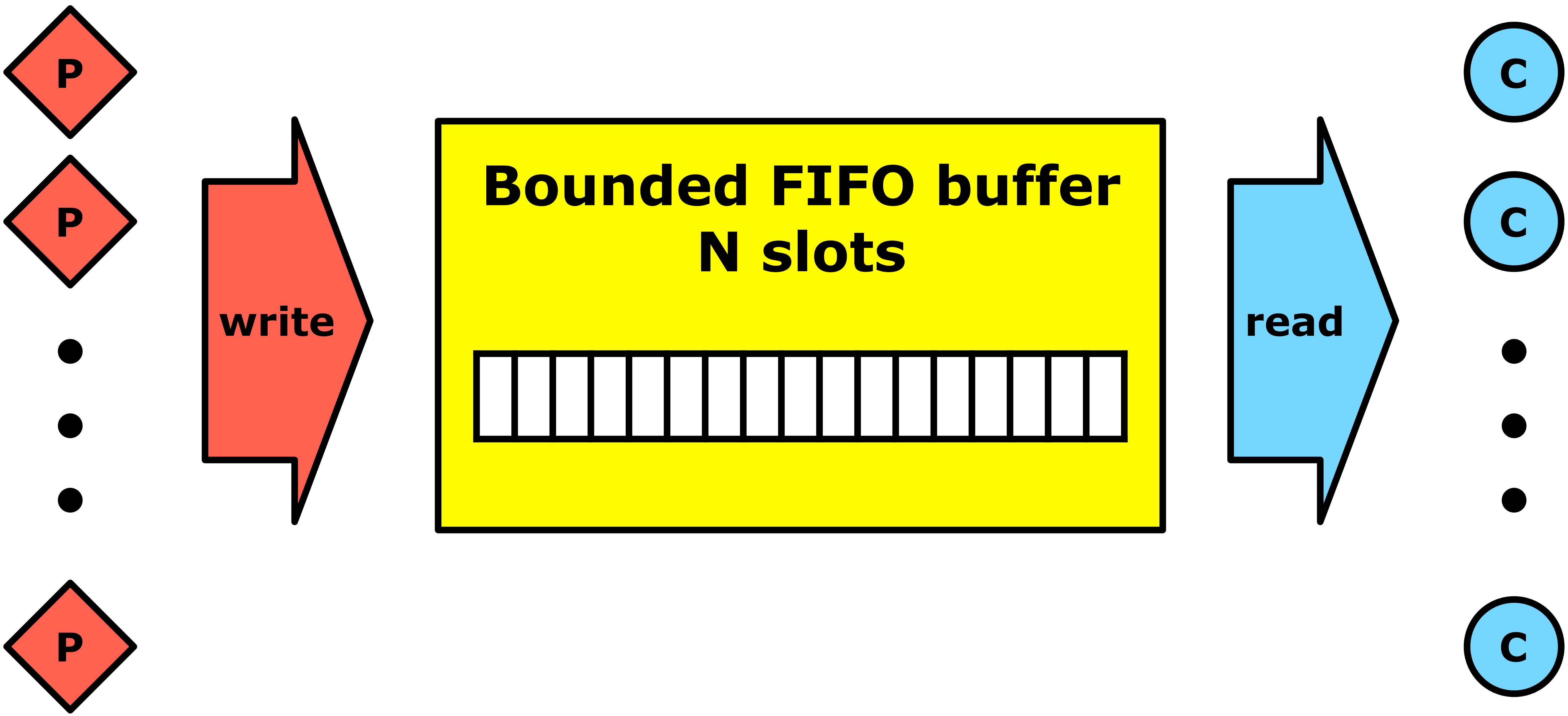
Classical problems of synchronization

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

Bounded buffer

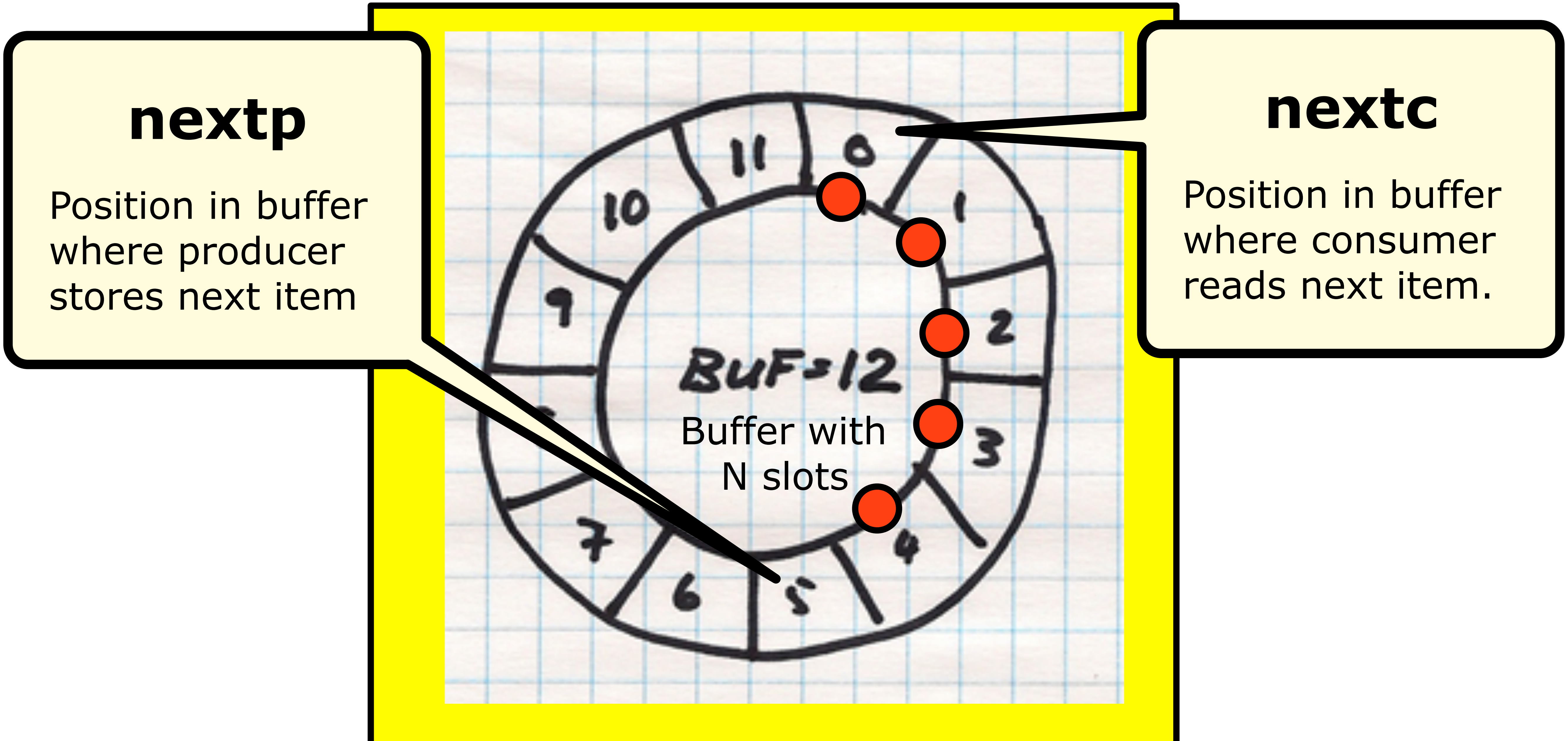
The bounded buffer problem

A set of **P** producers and **C** consumers using a shared bounded buffer of size **N**. Producers must block if the buffer is full. Consumers must block if buffer is empty.



The bounded buffer problem

A set of **P** producers and **C** consumers using a shared bounded buffer of size **N**. Producers must block if the buffer is full. Consumers must block if buffer is empty.

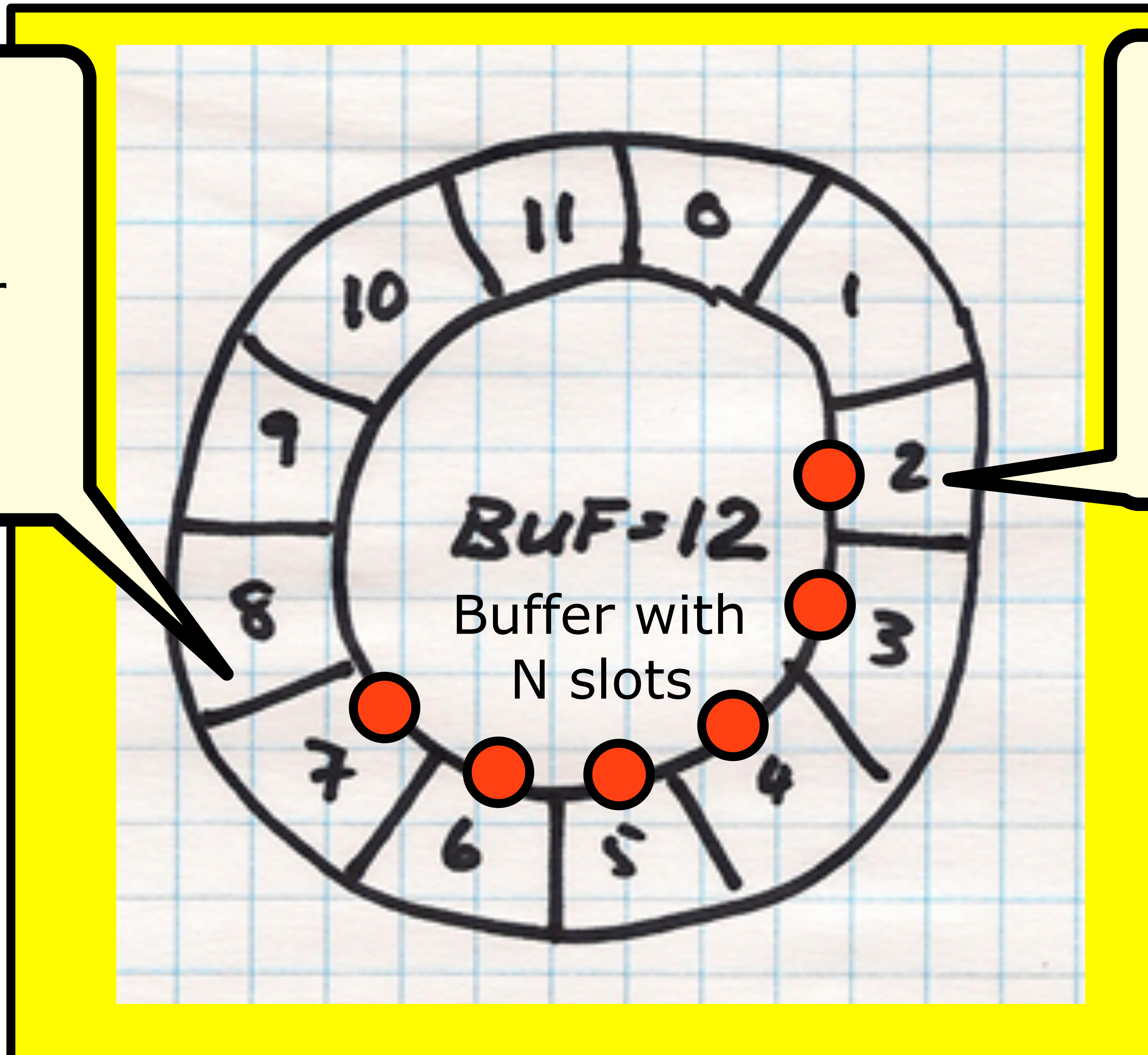


The bounded buffer problem

A set of **P** producers and **C** consumers using a shared bounded buffer of size **N**. Producers must block if the buffer is full. Consumers must block if buffer is empty.

nextp

Position in buffer where producer stores next item

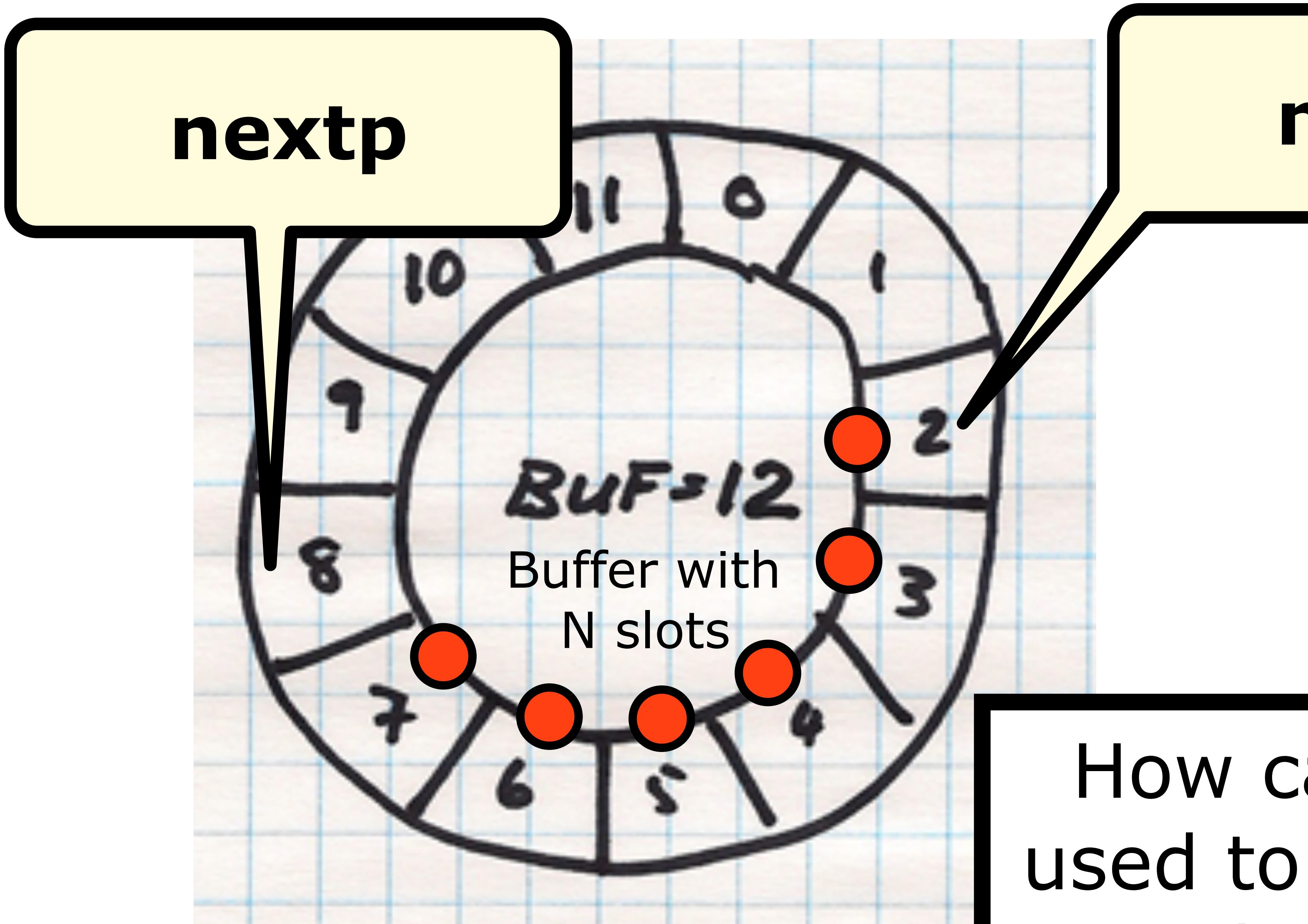


nextc

Position in buffer where consumer reads next item.

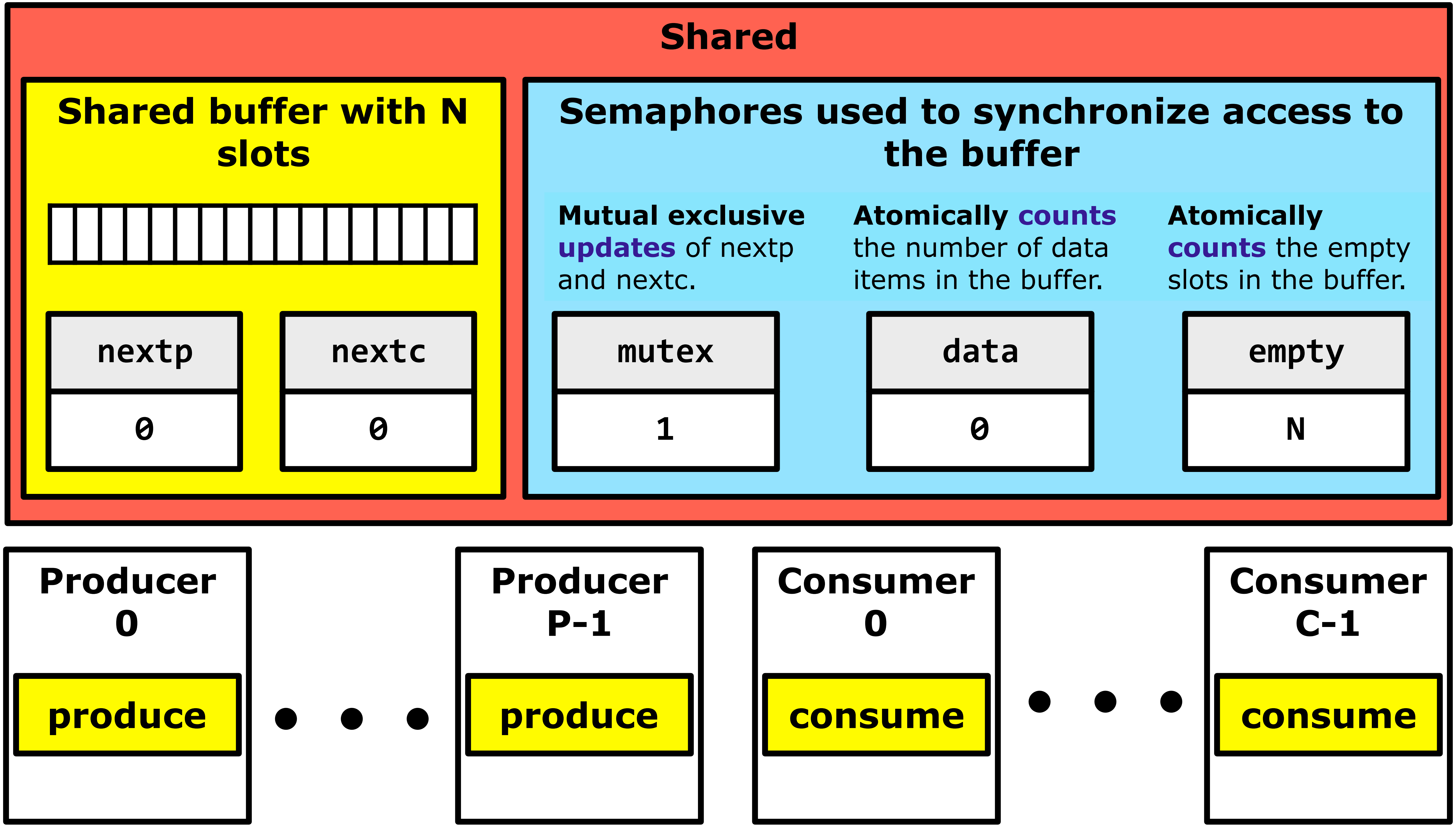
The bounded buffer problem

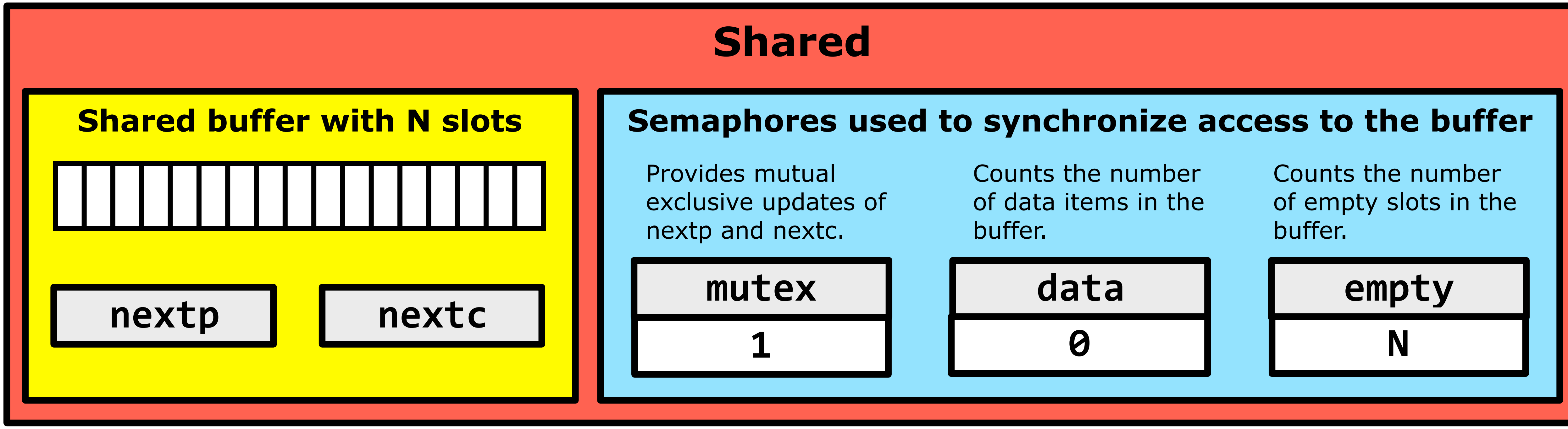
A set of **P** producers and **C** consumers using a shared bounded buffer of size **N**. Producers must block if the buffer is full. Consumers must block if buffer is empty.



How can semaphores be used to synchronize access to the bounded buffer?

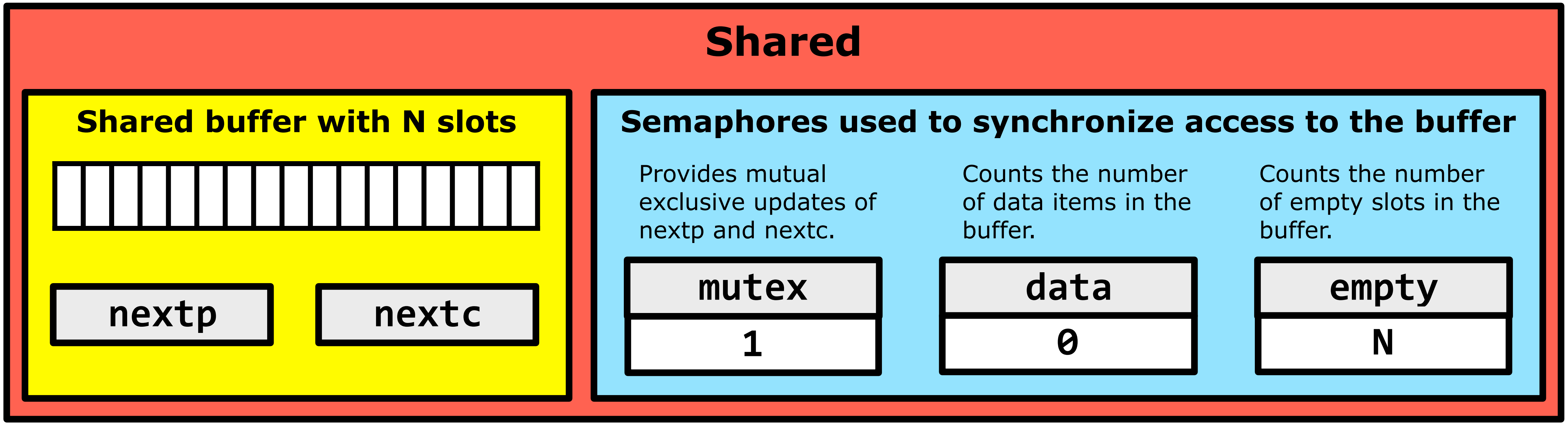
P producers and **C** consumers using a shared bounded buffer of size **N**. Producers writes to buffer at index **nextp** and consumer reads at index **nextc**.





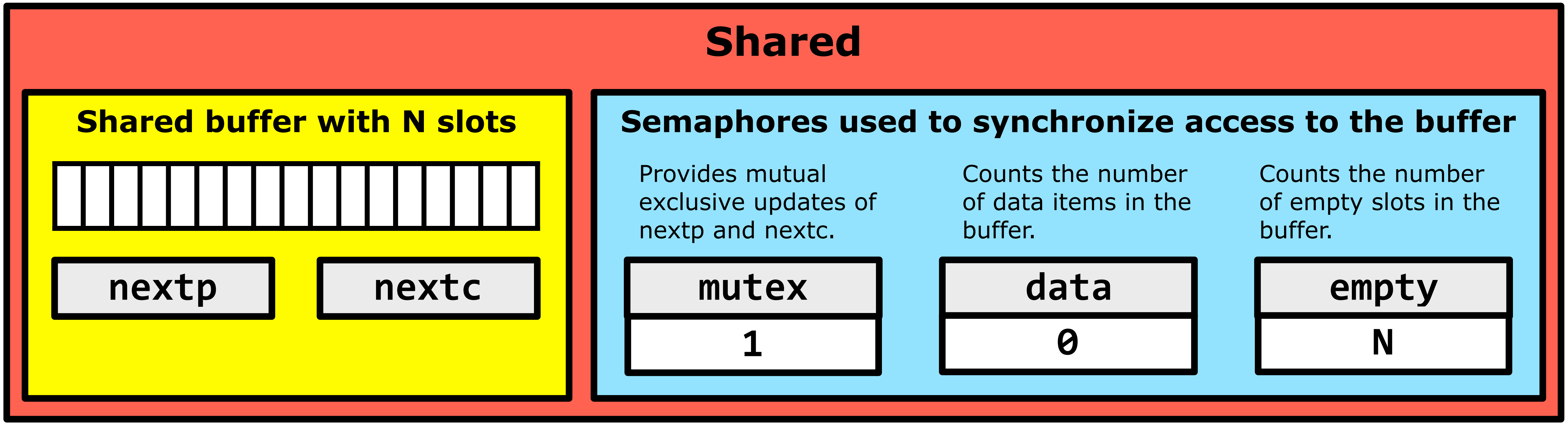

```
produce(buffer, *data) {
  wait(empty)
  wait(mutex)
  buffer[nextp] = copy(data)
  nextp = nexpt + 1 % N
  signal(mutex)
  signal(data)
}
```

1. **Block if buffer is full**, otherwise atomically decrement the empty item count.
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 nextp**.
5. **Leave the critical section**.
6. **Atomically increment the full item count**.




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

1. **Block if buffer is empty**, otherwise atomically decrement the full item count.
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 nextc**.
5. **Leave the critical section**.
6. **Atomically increment the empty item count**.




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

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

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

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

  data = copy(buffer[nextc])
  nextc = nextc + 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:

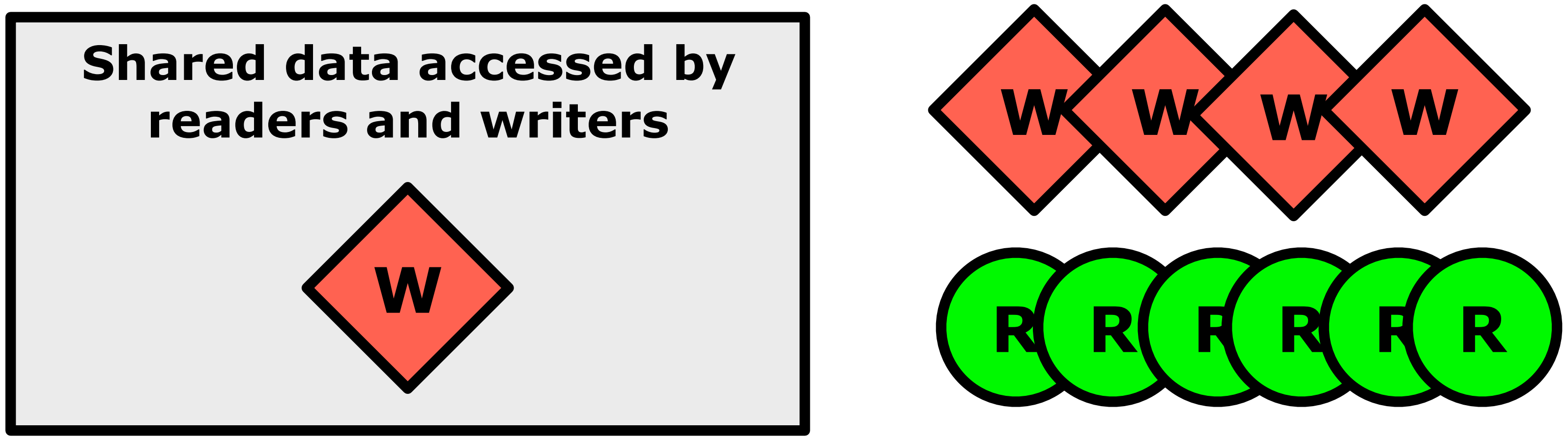
- `PIPE_BUF = 4096`.
- The value of `PIPE_BUF` is a consequence of other logic in the kernel, it is not a configuration parameter.

Readers and writers

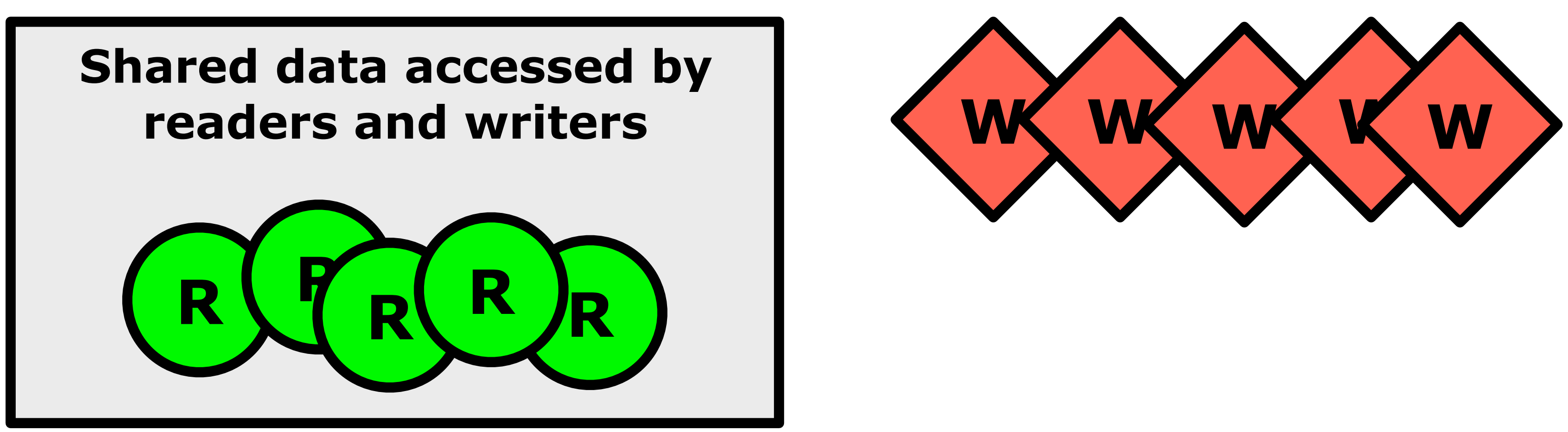
Readers-Writers Problem

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.



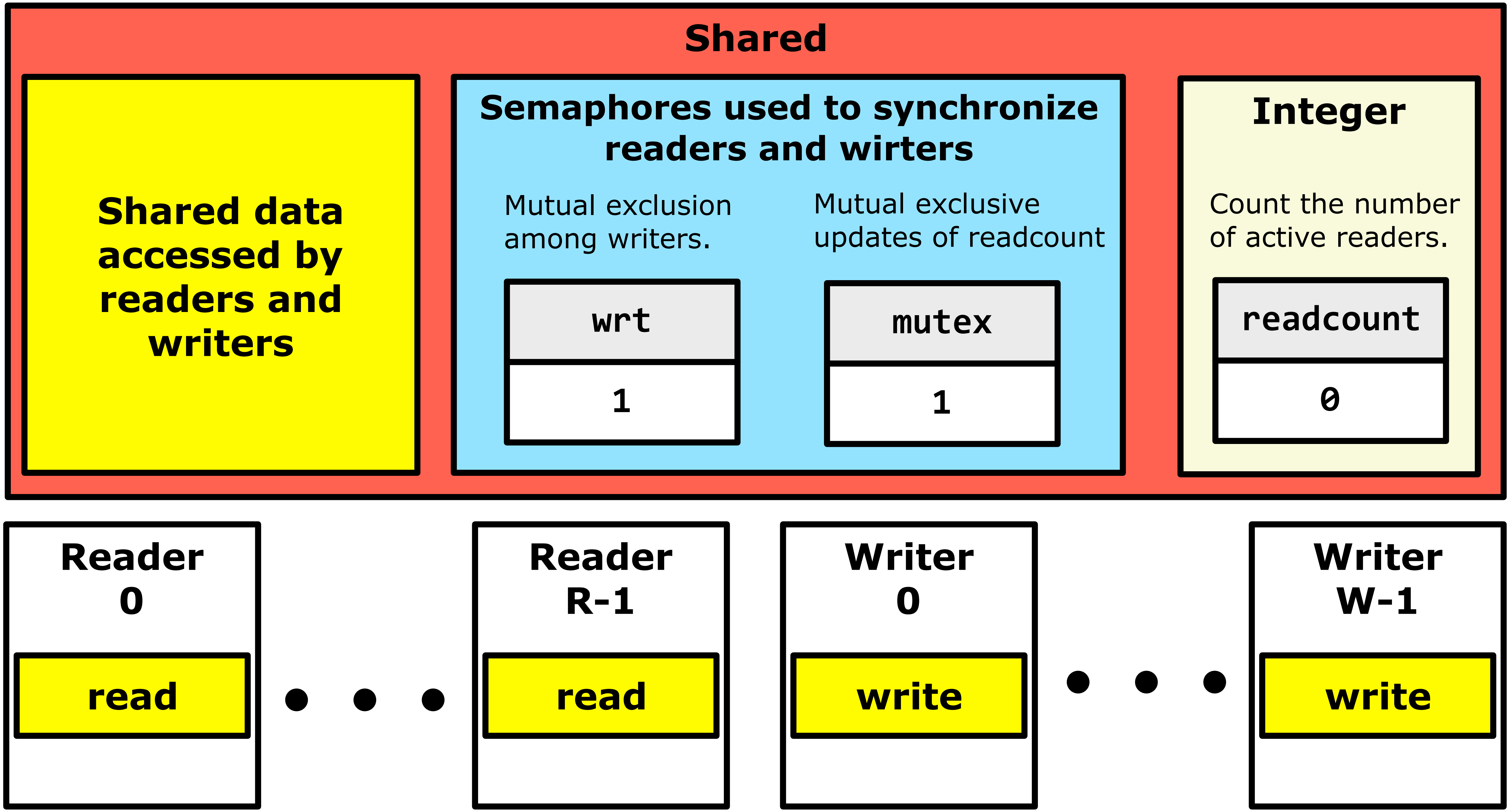
Allow multiple readers to read at the same time, any writers must be blocked

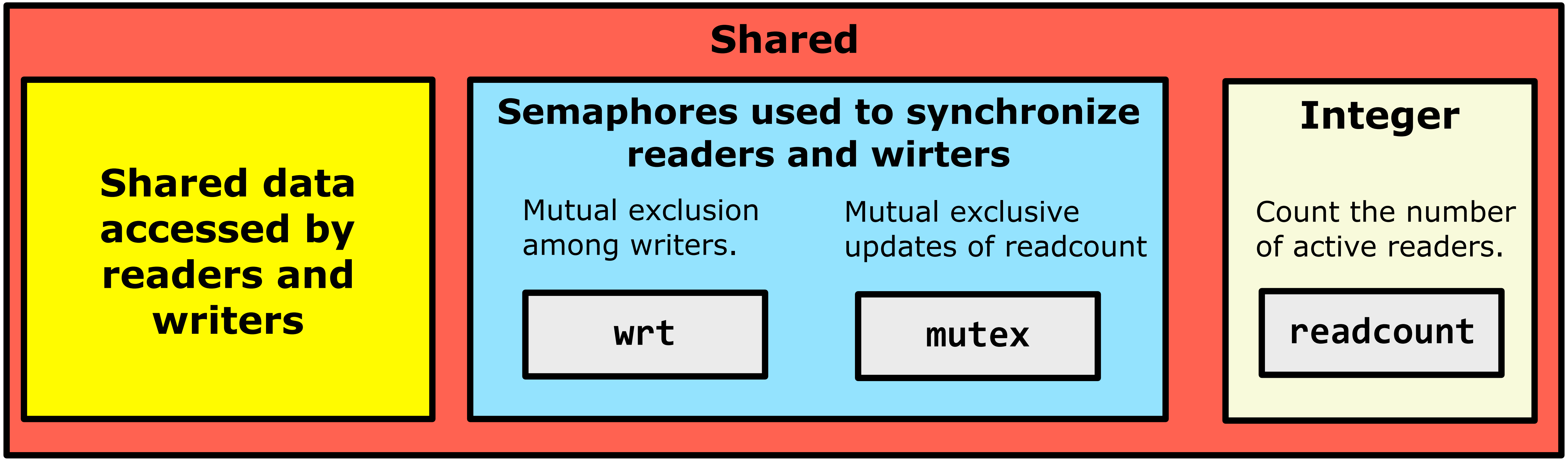


`R` readers and `W` writers access the same shared data set.

Allow multiple readers to read at the same time.

Only one single writer can access the shared data at the same time.






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

  // Write shared data

  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);

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

  signal(mutex);
}
```

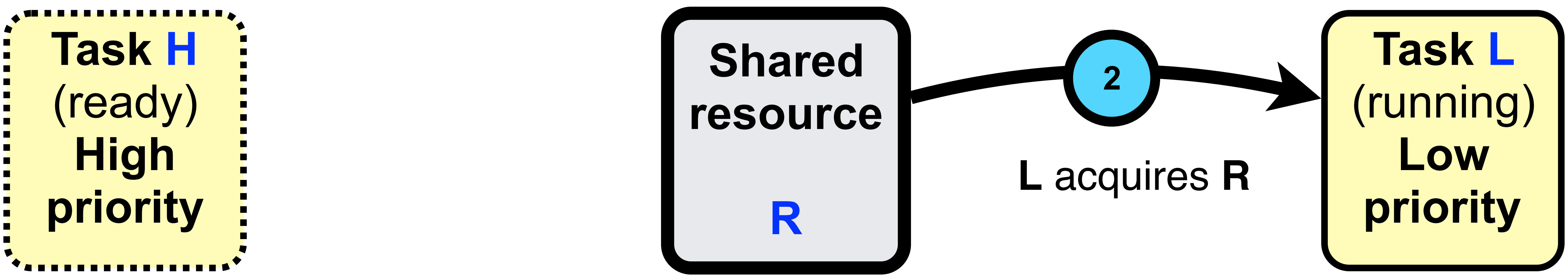
Priority inversion

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) **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.

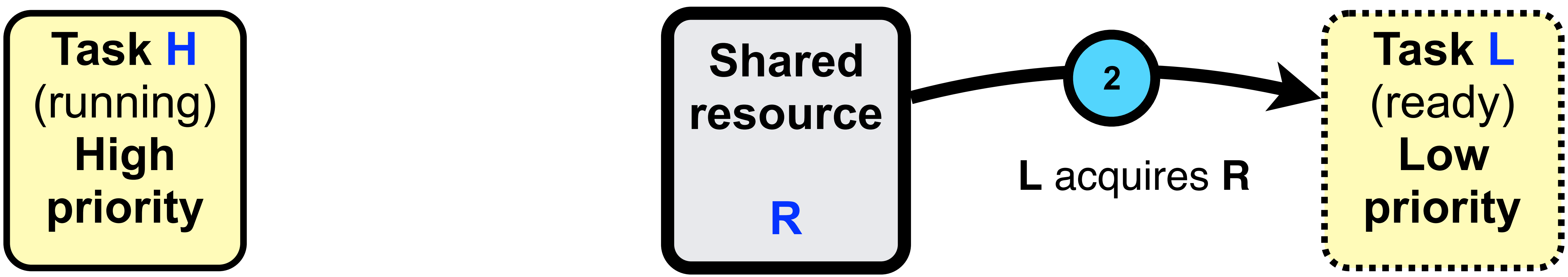


Scenario (1)

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

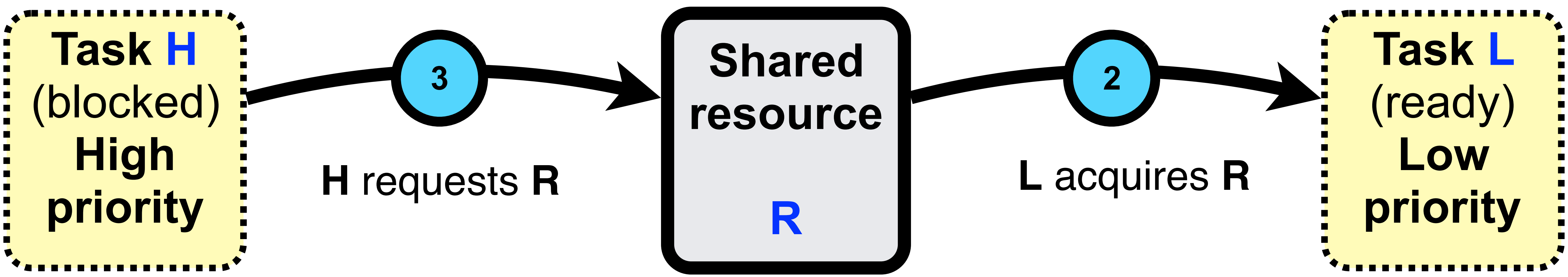
2) L acquires R.



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) **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.

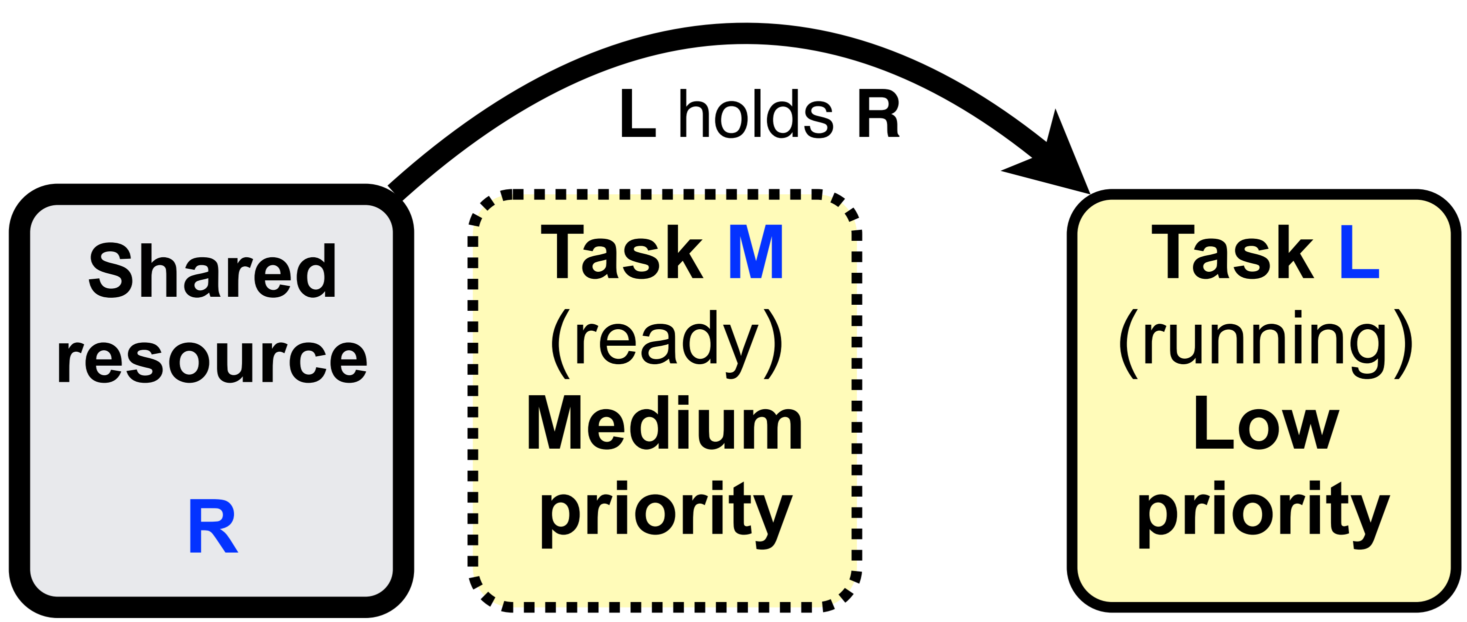


4) Sharing an exclusive-use resource (R in this case) in a well-designed system typically involves L relinquishing R promptly so that H (a higher priority task) does not stay blocked for excessive periods of time.

Scenario (2)

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

Task M of medium priority (p(L) < p(M) < p(H), where p(x) represents the priority for task x) becomes ready (to run) during L's use of R.



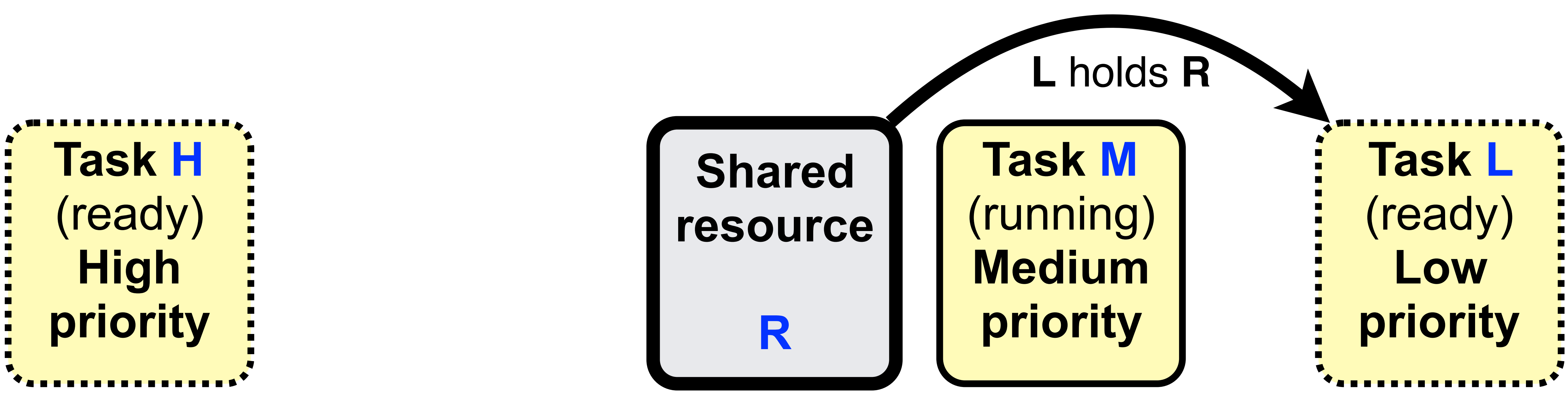
Scenario (2)

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 of medium priority (p(L) < p(M) < p(H), where p(x) represents the priority for task x) 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.

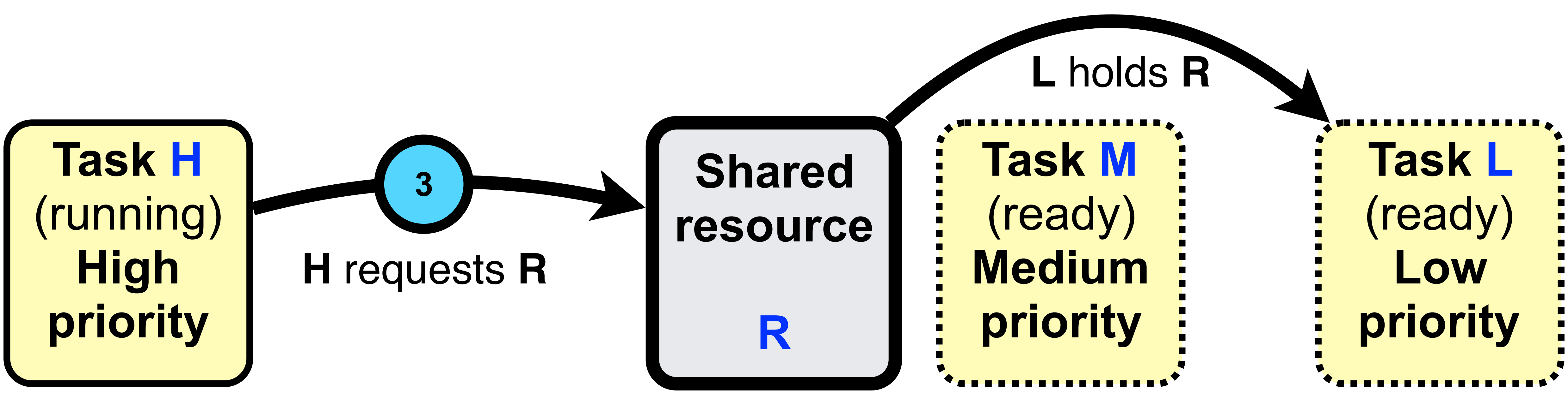


Scenario (2)

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 of medium priority (p(L) < p(M) < p(H), where p(x) represents the priority for task x) 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.

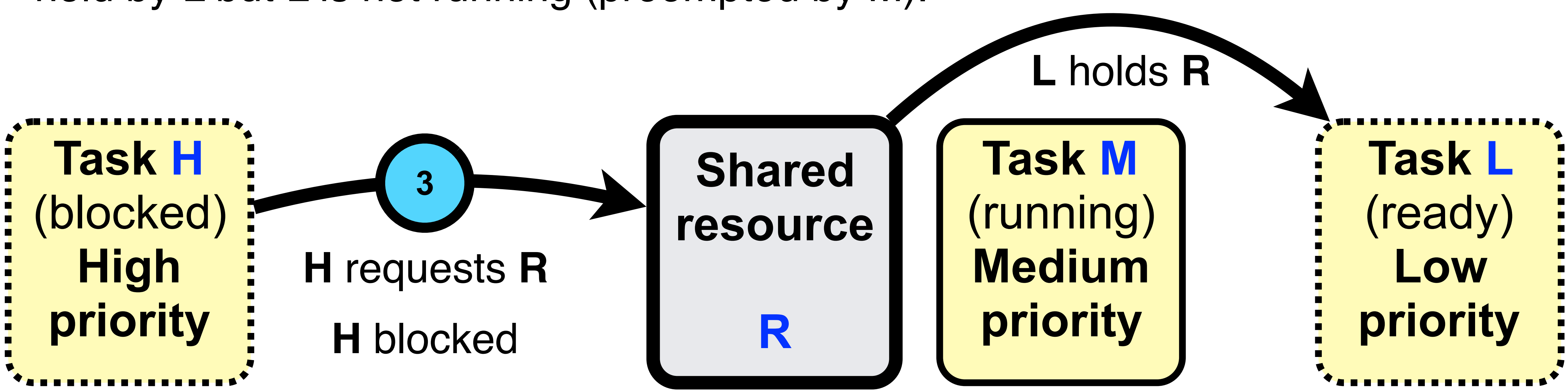


Scenario (2)

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 of medium priority (p(L) < p(M) < p(H), where p(x) represents the priority for task x) 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).

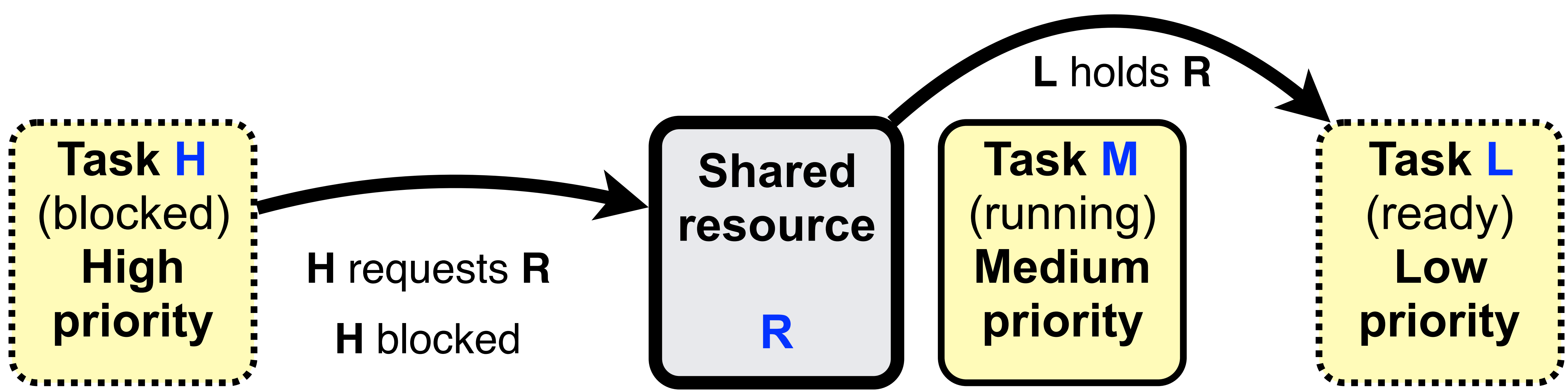


Priority inversion

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.