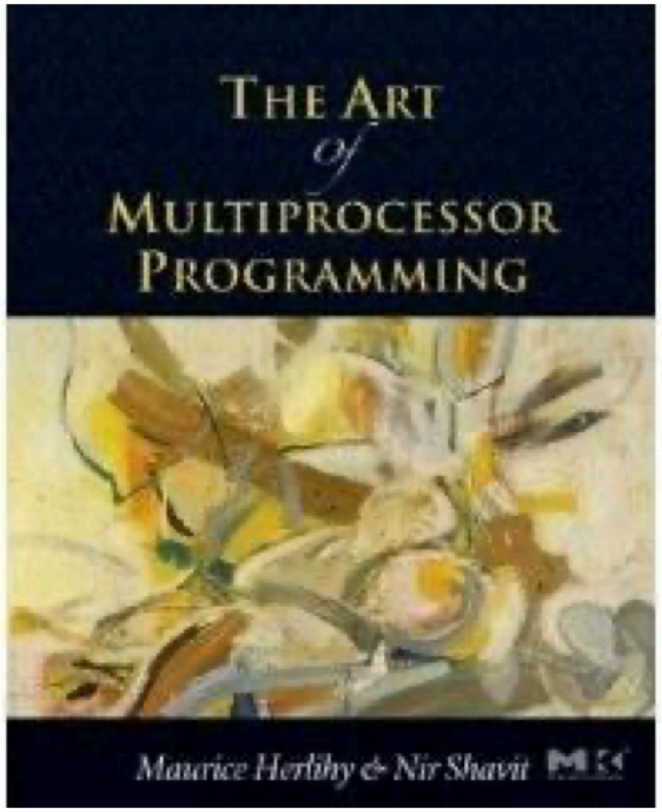
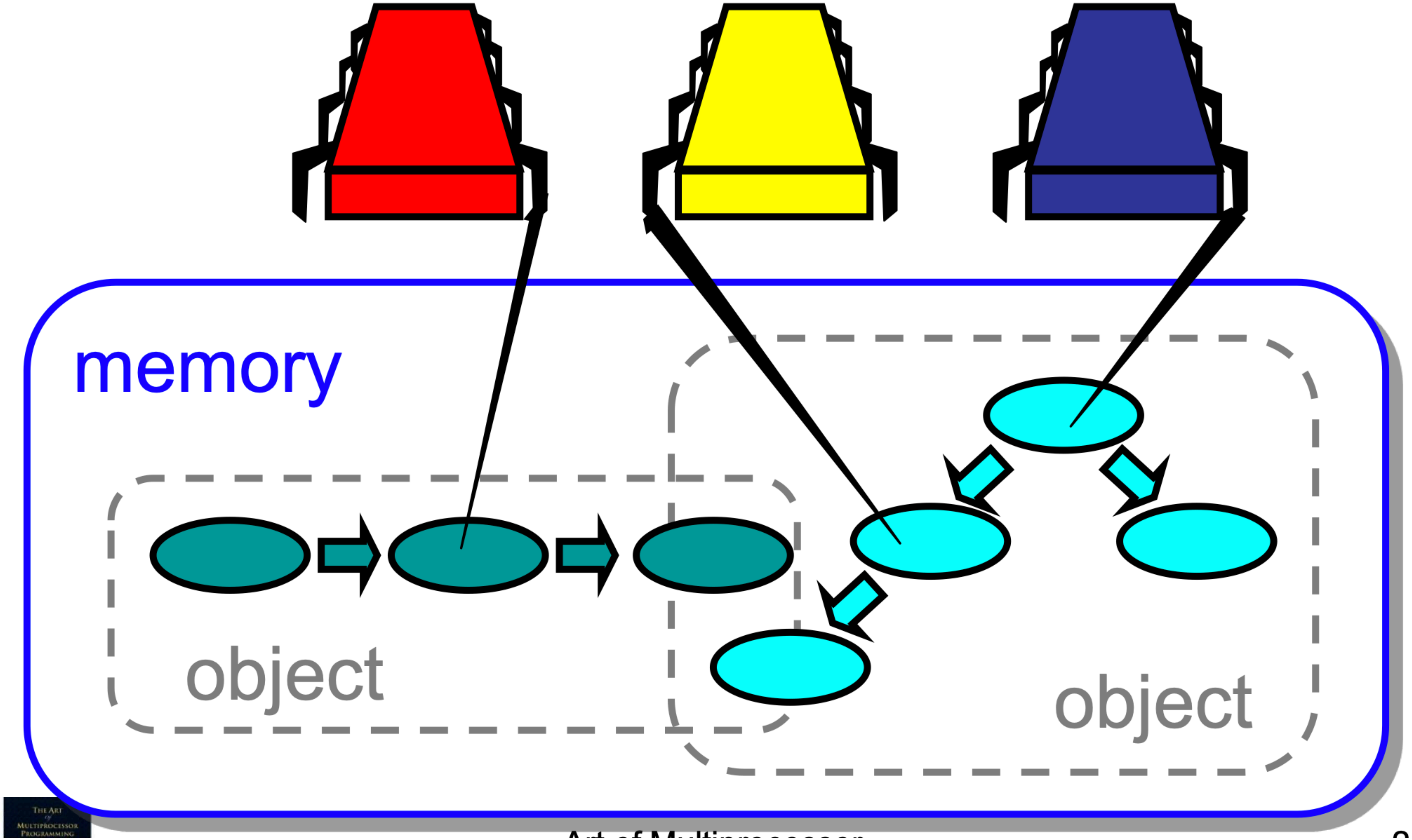
Concurrent Computation

Thomas Schwarz, SJ


THE ART *of* MULTIPROCESSOR PROGRAMMING

Maurice Herlihy & Nir Shavit

Concurrent Computation



memory

object

object

Concurrent Objects

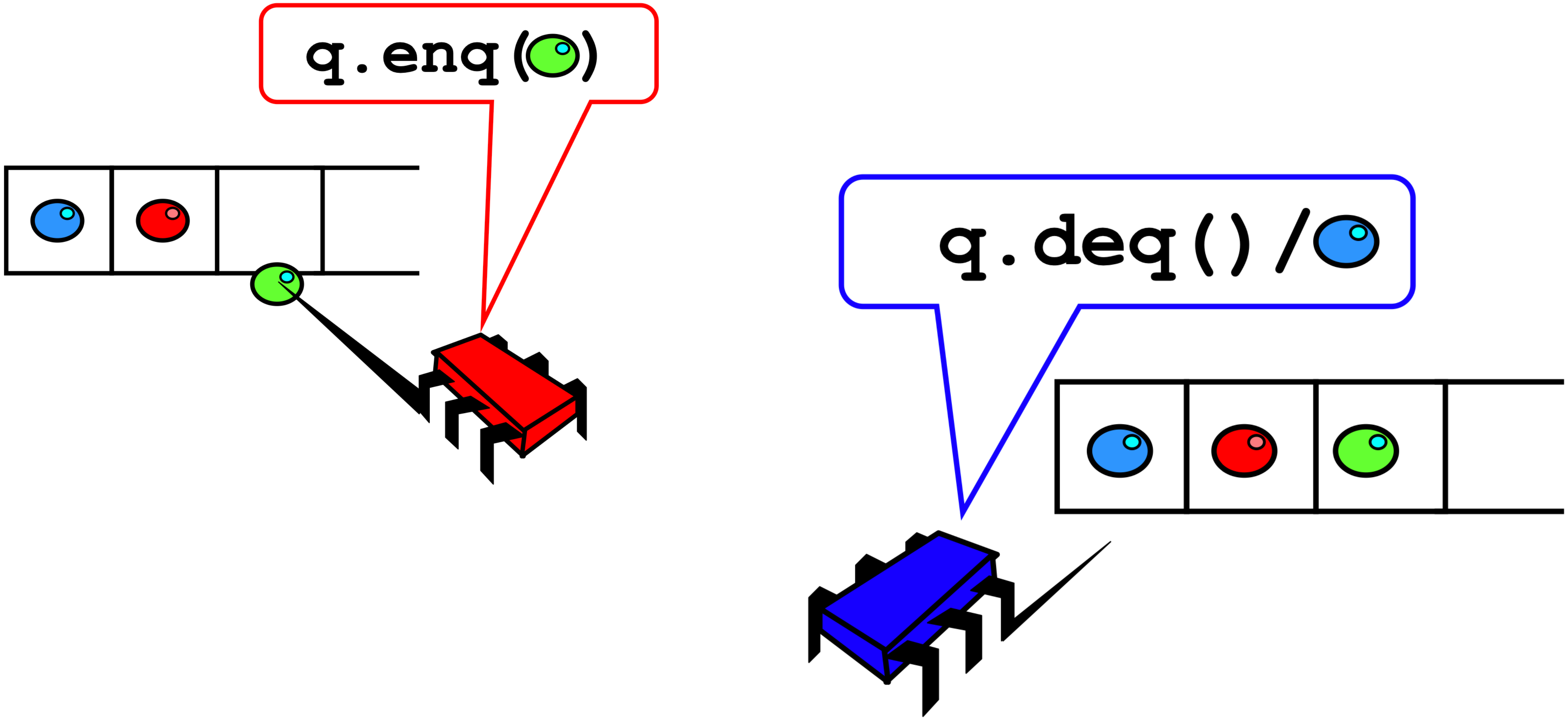
- What is a concurrent object?
  - How do we describe one
  - How do we implement one?
  - How do we tell if we are right?

Concurrent Objects

- Use
  - Safety (a.k.a. Correctness)
  - Liveness (a.k.a. progress)
- Base correctness on some equivalence with sequential behavior
  - We will look at:
    - Sequential consistency
    - Linearizability
    - Quiescent consistency
- Progress:
  - Blocking
  - Wait-free

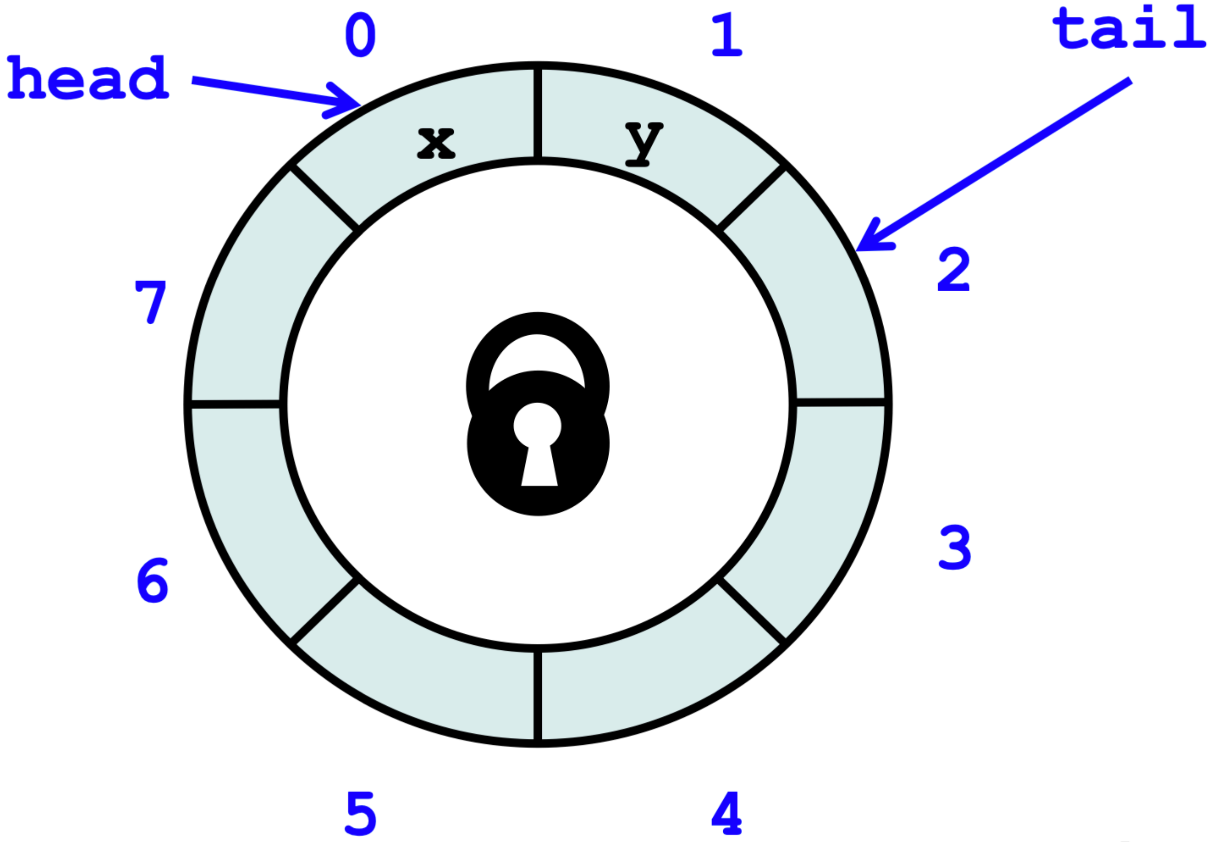
Example: FIFO Queue

- Insert at the tail, pop at the head



Lock-Based FIFO Queue

- Fields protected by a single, shared lock



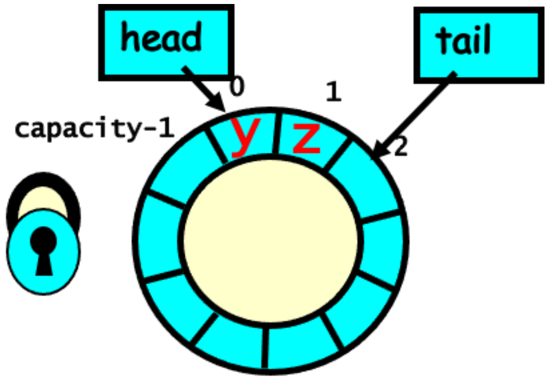
capacity = 8

```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

Lock-Based FIFO Queue

- Implementing DEQueue

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

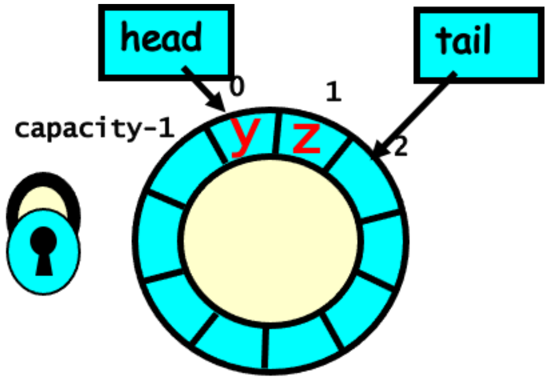


Lock-Based FIFO Queue

- Implementing DEQueue

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

only one operation at a time



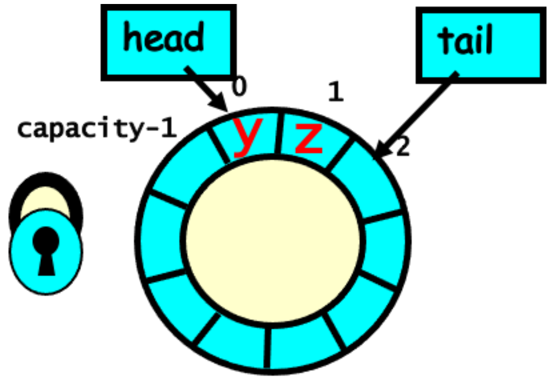
Lock-Based FIFO Queue

- Implementing DEQueue

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

if empty, throw exception

Lock will still be unlocked



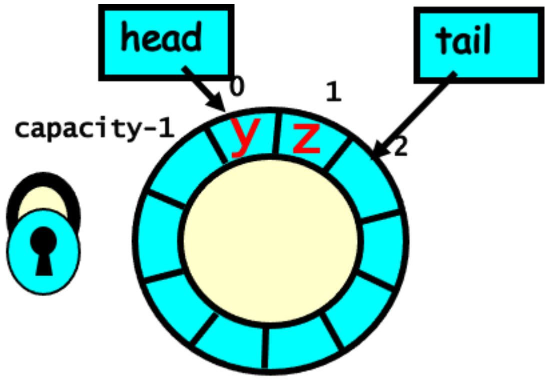
Lock-Based FIFO Queue

- Implementing DEQueue

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Queue not empty:

Remove first element
Reset head

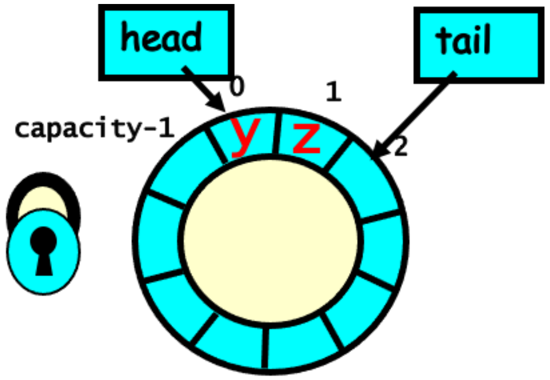


Lock-Based FIFO Queue

- Implementing DEQueue

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

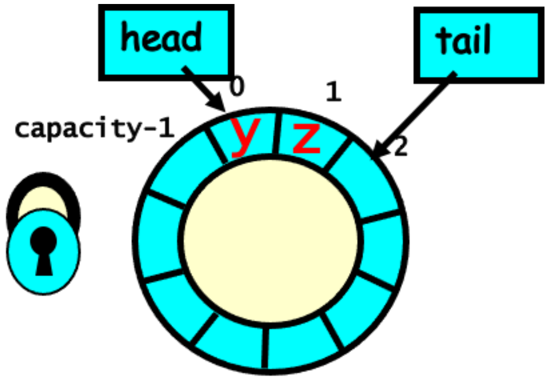
Return result



Lock-Based FIFO Queue

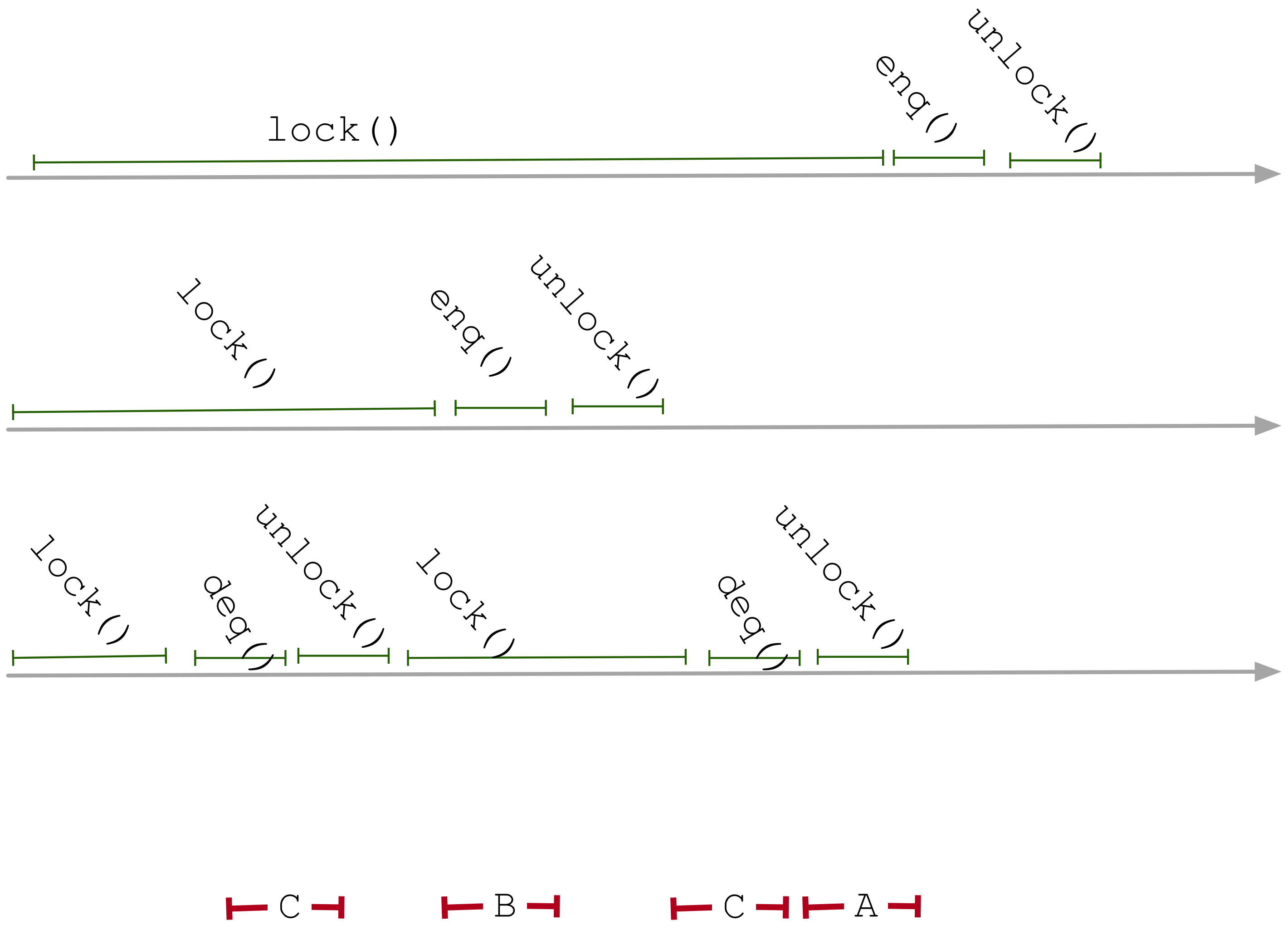
- Implementing DEQueue

```
public T enq() throws FullException{
  lock.lock();
  try {
    if (tail-1 == items.length)
       throw new FullException();
    items[tail % items.length]=x;
    tail++;
  } finally {
    lock.unlock();
  }
}
```



Lock-Based FIFO Queue

- Timeline
  - A enqueues
  - B enqueues
  - C dequeues
    - First time with empty exception
    - Second time returning B's insert



Lock Based FIFO Queue

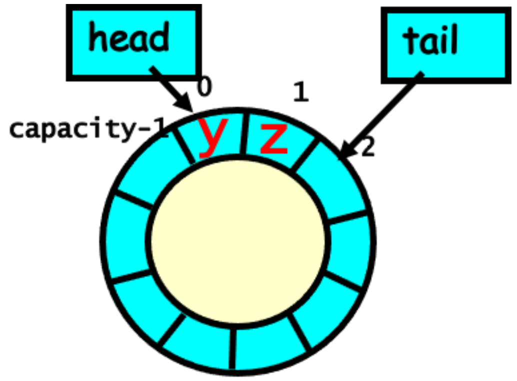
- Should be correct because concurrency is very limited

Wait-free 2-Thread Queue

- Mutual exclusion makes safety guarantees easy
  - But according to Amdahls law, has very little potential for speed-up
- Can build a wait-free queue, but only if there are only two threads:
  - One thread only enqueues
  - One thread only dequeues

Wait-free 2-Thread Queue

- Create cyclic queue as before



```
public class WaitFreeQueue {

  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

  public void enq(Item x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     while (tail == head);     // busy-wait
     Item item = items[head % capacity]; head++;
     return item;
}}
```

Wait-free 2-Thread Queue

```
public class WaitFreeQueue {

  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

  public void enq(Item x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     while (tail == head);     // busy-wait
     Item item = items[head % capacity]; head++;
     return item;
}}
```

Consensus

Thomas Schwarz, SJ

Consensus

- Consensus object has a single method
  - `int decide(int v)`
- Each thread calls decide exactly once
  - Output is
    - *consistent*: all threads decide on the same value
    - *valid:* the common decision value is some thread's input

Consensus

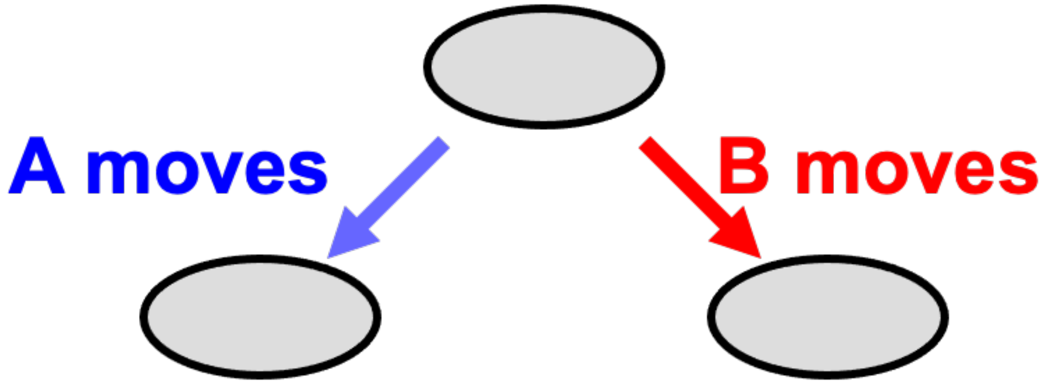
- A class *C* solves *n*-thread consensus if there exists a consensus protocol using any number of objects of class *C* and any number of atomic registers
- Consensus number *n* is the largest *n* for which that class solves *n*-thread consensus

Binary Consensus

- Two threads decide on 0 or 1

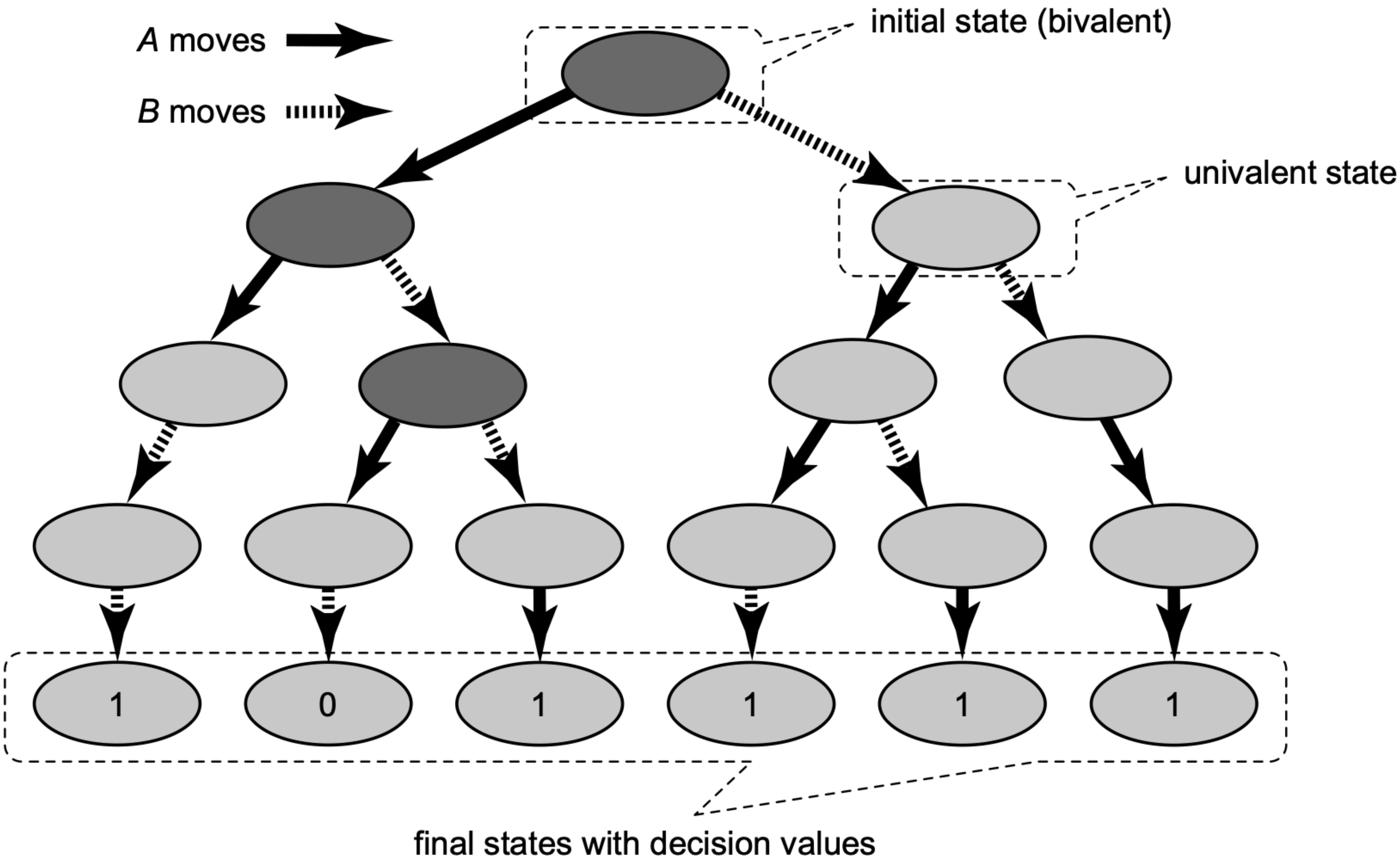
Binary Consensus

- Model execution with a tree model of state transitions



- A state is called univalent if all children decide on the same value
- A state is called bivalent otherwise

Binary Consensus



Binary Consensus

- Lemma 1: Every 2-thread consensus protocol has a bivalent initial state
- Proof:
    - Initial state: A has input 0 and B has input 1
    - If A finishes the protocol before B takes a step, then A must decide on 0, because this is the only input it has seen
    - If B finishes the protocol before A takes a step, then A must decide on 1
    - It follows that the initial state where A has 0 and B has 1 is bivalent

Binary Consensus

- Lemma 2: Every *n*-thread consensus protocol has a bivalent initial state
  - Homework 3

Binary Consensus

- A protocol state is *critical if*
  - It is bivalent
  - If any thread moves, the protocol state becomes univalent

Binary Consensus

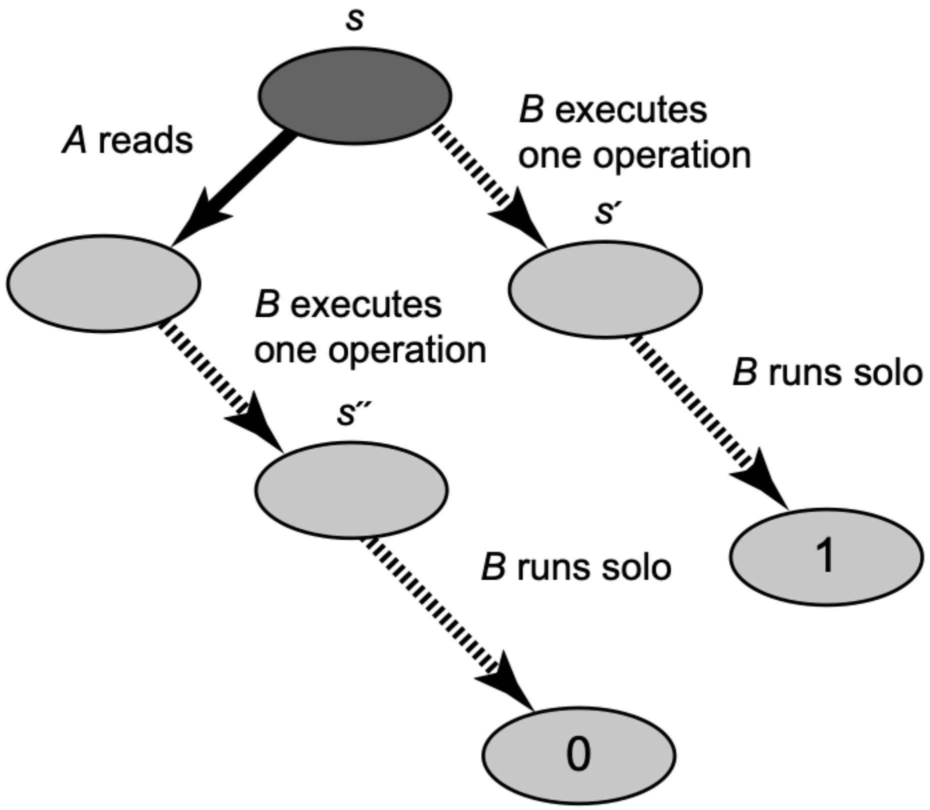
- Lemma 3: Every wait-free consensus protocol has a critical state
  - Proof: Suppose not.
  - The protocol has a bivalent initial state.
  - As long as there is thread that can move without making the state univalent, let this thread move
  - If the protocol runs for-ever, then it is not wait-free
  - Otherwise, the protocol eventually enters a state where no such move is possible, which is a critical state

Binary Consensus

- Atomic registers have consensus number 1
  - Suppose there is a binary consensus protocol for two threads A and B
  - By Lemma 3, the protocol reaches a critical state *s*
    - *WLOG*: A's next move carries the protocol to a 0-valent state and B's next move to a 1-valent state

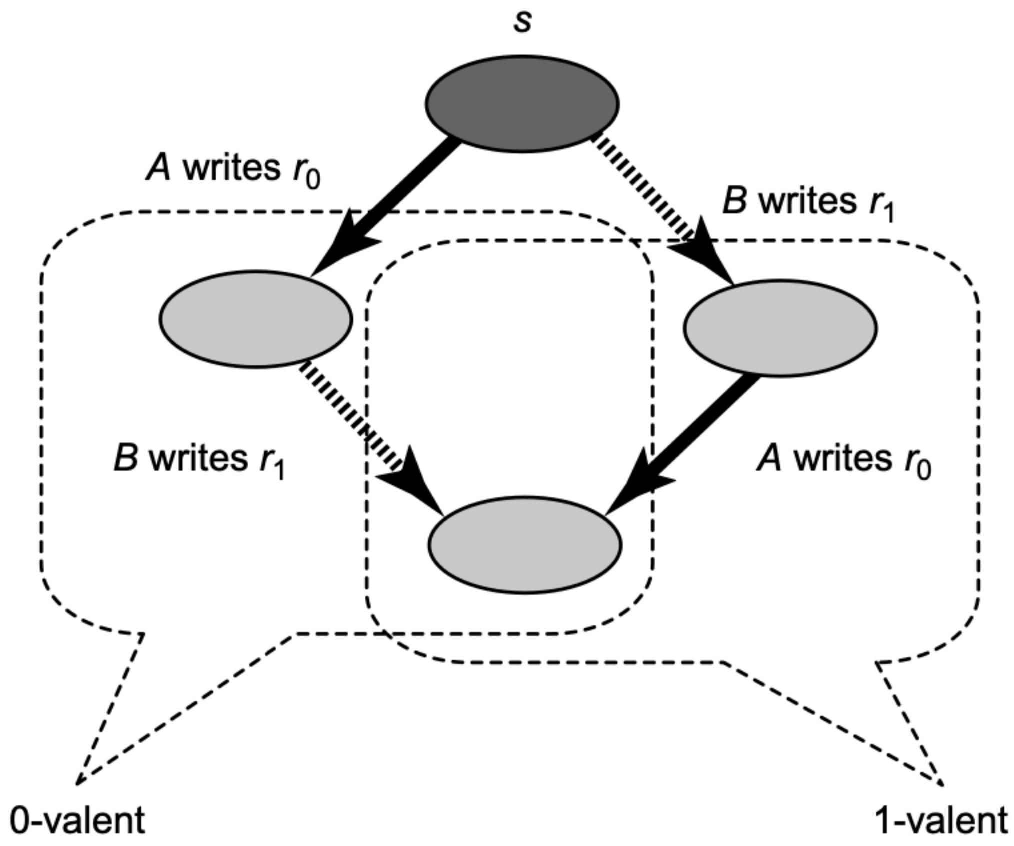
Binary Consensus

- Case 1: A reads a certain register
- Scenario 1: B moves first
  - Drives protocol to a 1-valent state
  - Then runs solo
- Scenario 2: A moves first driving protocol to a 0-valent state.
  - B then moves and runs solo
- But States s' and s'' are undistinguishable for B, so they should have the same outcome



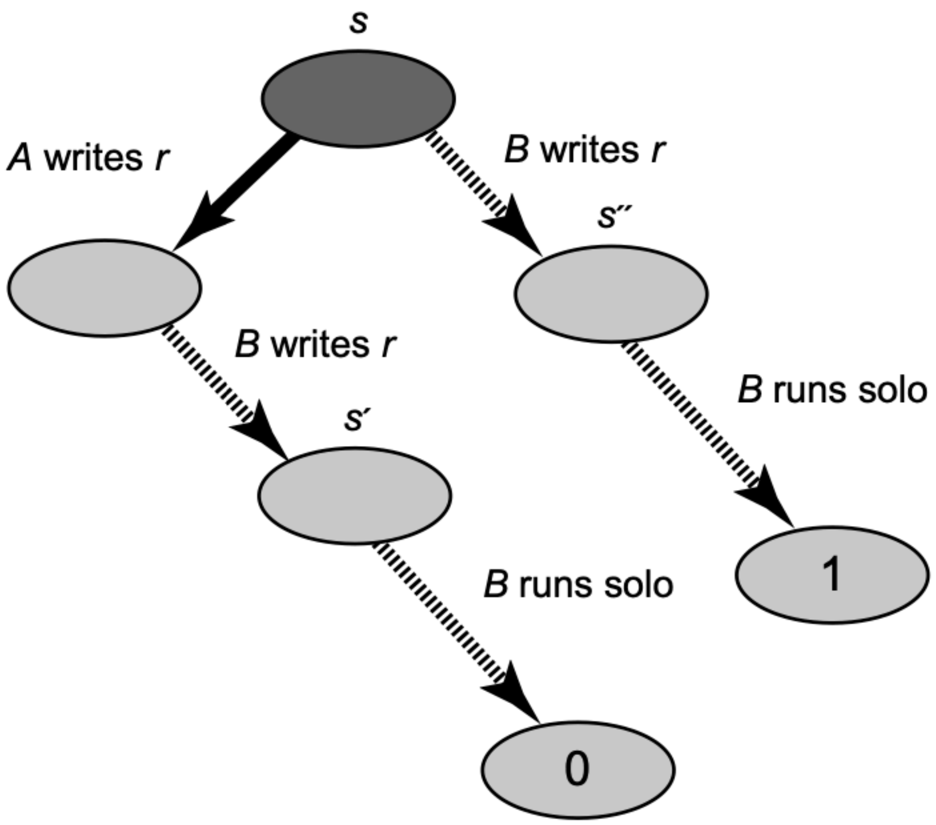
Binary Consensus

- In the critical state:
- Both write to different registers r_0 and r_1
- If A moves first, then we go to a 0-valent state
- If B moves first, then we go to a 1-valent state.
- But if the other then writes their register, we have the same state, which is therefore both 0- and 1-valent



Binary Consensus

- Remaining case:
  - A and B write to the same register
  - Scenario 1: A writes and then B runs solo: 0-valent
  - Scenario 2: B writes and then runs solo: 1-valent
  - But states s' and s'' are indistinguishable



Binary Consensus

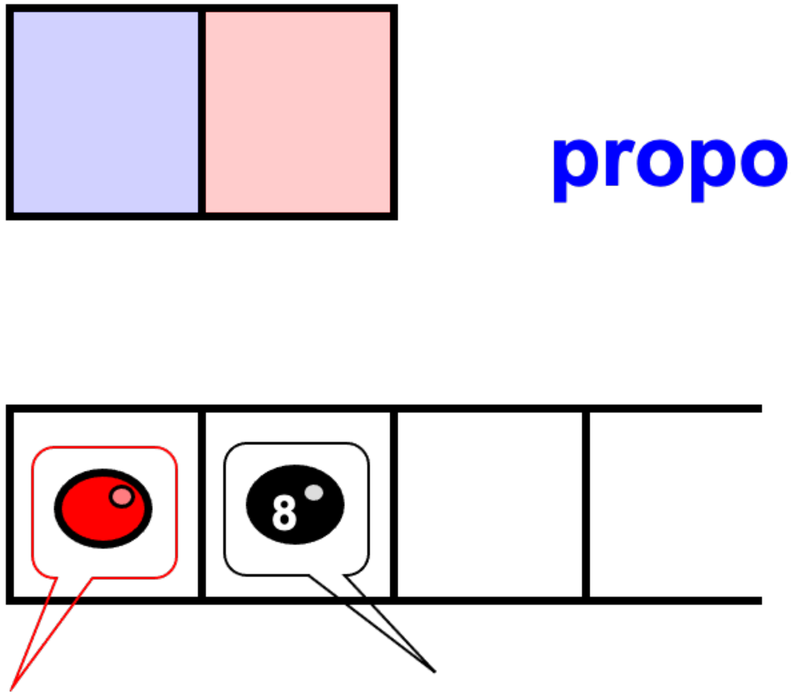
- Impossible to construct a wait-free consensus protocol with atomic registers only

FIFO Queues

- Previously: wait-free FIFO queue using only atomic registers
  - **AS LONG AS** one enqueuer thread and one dequeuer thread
- Assume that we have a wait-free FIFO queue with two dequeuers
-

FIFO Queues

- 2-Dequeuer FIFO Queue solves 2-thread consensus
    - Idea: Place a WIN and a LOOSE value into the queue

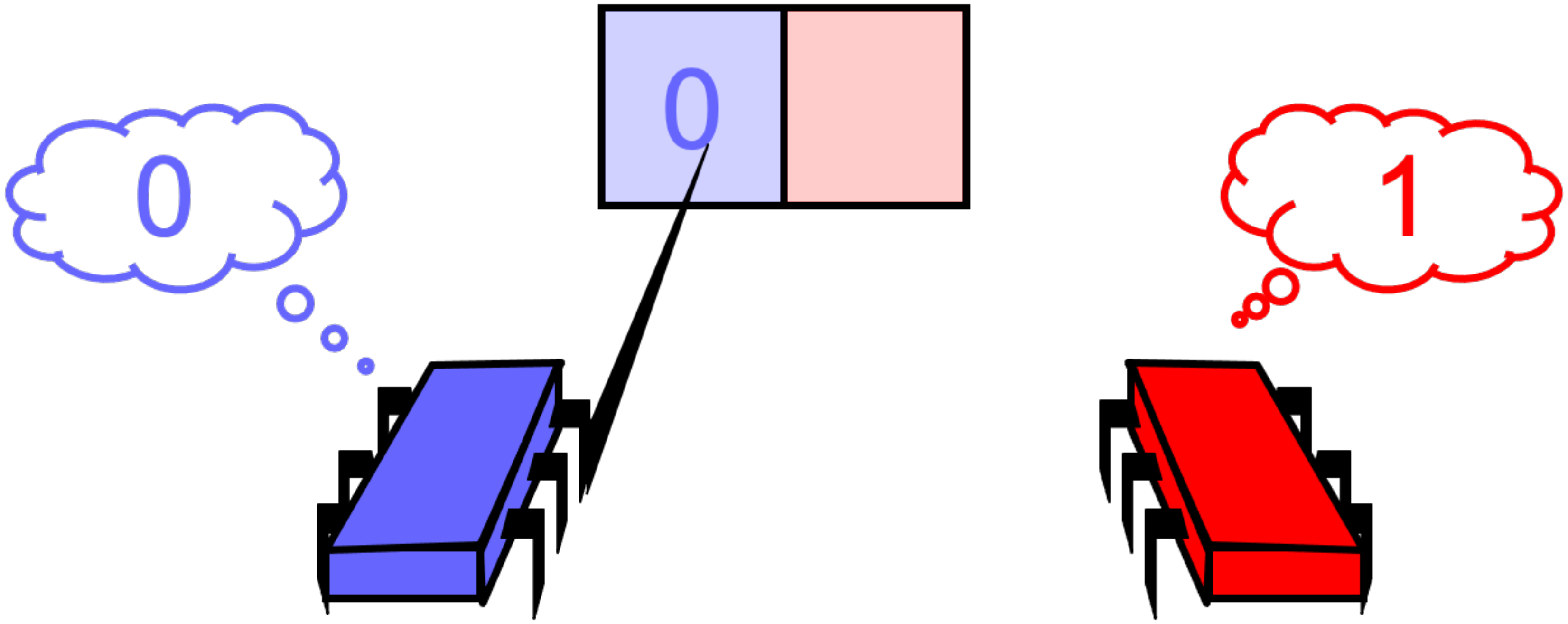


proposed array

FIFO Queue with red and black balls

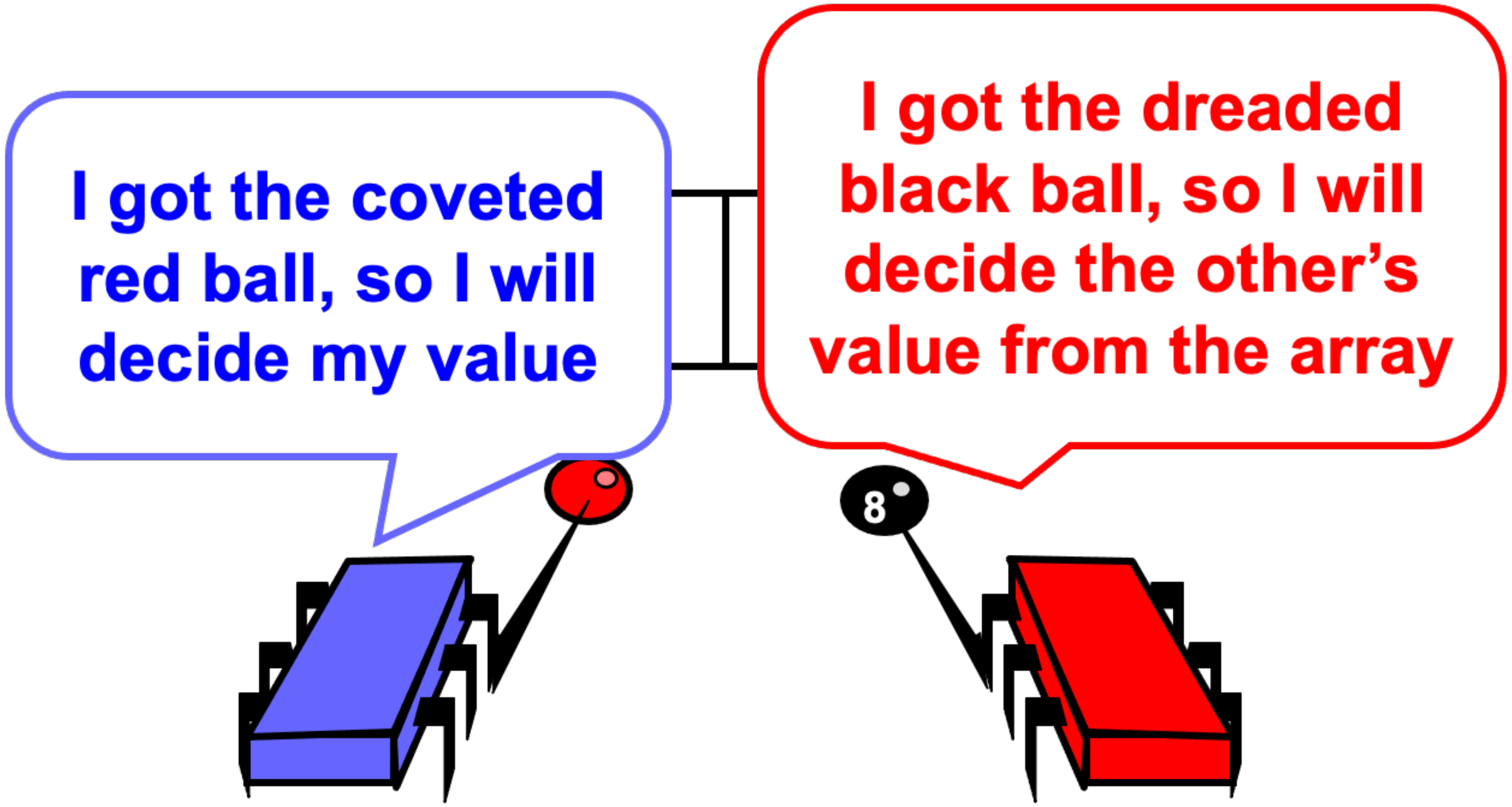
FIFO Queues

- Each thread writes a value to the array



FIFO Queues

- Each thread takes an item from the queue



FIFO Queues

```
public class QueueConsensus<T> extends ConsensusProtocol<T> {
private static final int WIN = 0; // first thread
private static final int LOSE = 1; // second thread

Queue queue;
// initialize queue with two items
public QueueConsensus() {
   queue = new Queue();
   queue.enq(WIN);
   queue.enq(LOSE);  }
//figure out which thread was first
 public T decide(T Value) {
     propose(value);
     int status = queue.deq();
     int i = ThreadID.get();
     if (status == WIN)
         return proposed[i];
     else
         return proposed[1-i];
     }
}
```

FIFO Queues

- Correctness:
  - One thread gets the red ball
  - The other thread gets the black ball
  - Winner decides on their own value
  - Looser can find winner's value in the array

FIFO Queues

- Therefore:
  - It is impossible to implement a wait-free two dequeuer FIFO queue from atomic registers

FIFO Queues

- FIFO queues cannot solve three-consensus