

# **04 Monitors and Blocking Synchronisation**

**CS 6868: Concurrent Programming**

KC Sivaramakrishnan

**Spring 2026, IIT Madras**

# Our Focus

- **Keep trying**
  - “spin” or “busy-wait”
  - Good if delays are short
- **Give up the processor**
  - Good if delays are long
  - Always good on uniprocessor

*Previous lecture*

*This lecture*

# Lock-based queue (from lecture 3)

```
exception Full
exception Empty
```

```
type 'a t = {
  items : 'a option array;
  capacity : int;
  mutable head : int;
  mutable tail : int;
  lock : Mutex.t;
}
```

```
let create capacity =
{
  items = Array.make capacity None;
  capacity;
  head = 0;
  tail = 0;
  lock = Mutex.create ();
}
```

```
let deq q =
  Mutex.lock q.lock;
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)
  (fun () ->
```

```
    if q.tail = q.head then
      raise Empty;
```

```
    match q.items.(q.head mod q.capacity) with
    | None ->
      assert false (* Should never happen *)
    | Some x ->
      q.items.(q.head mod q.capacity) <- None;
      q.head <- q.head + 1;
      x)
```

*deq raises **Empty** if the queue is empty*

# Lock-based queue (from lecture 3)

```
exception Full
exception Empty
```

```
type 'a t = {
  items : 'a option array;
  capacity : int;
  mutable head : int;
  mutable tail : int;
  lock : Mutex.t;
}
```

```
let create capacity =
{
  items = Array.make capacity None;
  capacity;
  head = 0;
  tail = 0;
  lock = Mutex.create ();
}
```

```
let deq q =
  Mutex.lock q.lock;
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)
    (fun () ->
```

```
    if q.tail = q.head then
      raise Empty;
```

```
    match q.items.(q.head mod q.capacity) with
    | None ->
      assert false (* Should never happen *)
    | Some x ->
      q.items.(q.head mod q.capacity) <- None;
      q.head <- q.head + 1;
      x)
```

*deq raises **Empty** if the queue is empty*

*How to wait till there is an element?*

# Spin-wait

```
exception Full
exception Empty
```

```
type 'a t = {
  items : 'a option array;
  capacity : int;
  mutable head : int;
  mutable tail : int;
  lock : Mutex.t;
}
```

```
let create capacity =
{
  items = Array.make capacity None;
  capacity;
  head = 0;
  tail = 0;
  lock = Mutex.create ();
}
```

```
let defq q =
  Mutex.lock q.lock;
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)
  (fun () ->
```

```
    while q.tail = q.head do
      Mutex.unlock q.lock;
      Domain.cpu_relax ();
      Mutex.lock q.lock
    done;
```

```
    match q.items.(q.head mod q.capacity) with
    | None -> assert false (* Should never happen *)
    | Some x ->
      q.items.(q.head mod q.capacity) <- None;
      q.head <- q.head + 1;
      x)
```

*Allow the **enq** operation to interleave*

# Downsides of Spin-wait

- Downside of spin-wait
  - Actively wasting CPU spin-waiting
    - On a multi-processor system, there may be other useful tasks to do
  - Waste a full time slice on a uniprocessor system
    - Assuming the OS does preemptive multi-threading every time slice
- What if we expect to wait for significant time?
  - Better to **block** until element is available in the queue.

# Condition Variables

- Temporarily give up a critical section and **block**
  - Goes to sleep with the help of the OS; not using CPU actively
- Maybe signalled later to **wake up** the waiting threads
- Woken up thread **resumes** in the critical section
- Always associated with a **Mutex**
- Monitors = Mutex + Conditional Variables
  - Introduced in 1973 paper by Sir Tony Hoare
  - Same person who invented QuickSort and Program Logics

# Condition Variables

```
(** Condition variables *)
```

```
type t
```

```
(** [create ()] creates and returns a new condition variable. *)
```

```
val create : unit -> t
```

```
(** [wait c m] atomically unlocks the mutex [m] and suspends the  
calling thread on the condition variable [c]. The thread will  
resume after being woken up via [signal] or [broadcast], at  
which point the mutex [m] is locked again before [wait] returns. *)
```

```
val wait : t -> Mutex.t -> unit
```

```
(** [signal c] wakes up one of the threads waiting on the condition  
variable [c], if there is one. If there are no threads waiting  
on [c], this call has no effect. *)
```

```
val signal : t -> unit
```

```
(** [broadcast c] wakes up all threads waiting on the condition  
variable [c]. If there are no threads waiting on [c], this  
call has no effect. *)
```

```
val broadcast : t -> unit
```



# Blocking Queue

```
type 'a t = {  
  items : 'a option array;  
  capacity : int;  
  mutable head : int;  
  mutable tail : int;  
  lock : Mutex.t;  
  not_empty : Condition.t; (* Signaled when queue becomes non-empty *)  
  not_full : Condition.t;  (* Signaled when queue becomes non-full *)  
}
```

```
let create capacity =  
  {  
    items = Array.make capacity None;  
    capacity;  
    head = 0;  
    tail = 0;  
    lock = Mutex.create ();  
    not_empty = Condition.create ();  
    not_full = Condition.create ();  
  }
```

# Blocking Queue

```
let def q =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is empty *)  
    while q.tail = q.head do  
      Condition.wait q.not_empty q.lock  
    done;  
  
    match q.items.(q.head mod q.capacity) with  
    | None -> assert false (* Should never happen *)  
    | Some x ->  
      q.items.(q.head mod q.capacity) <- None;  
      q.head <- q.head + 1;  
      (* Signal that queue is not full *)  
      Condition.signal q.not_full;  
      x)
```

# Blocking Queue

```
let def q =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is empty *)  
    while q.tail = q.head do  
      Condition.wait q.not_empty q.lock  
    done;  
  
    match q.items.(q.head mod q.capacity) with  
    | None -> assert false (* Should never happen *)  
    | Some x ->  
      q.items.(q.head mod q.capacity) <- None;  
      q.head <- q.head + 1;  
      (* Signal that queue is not full *)  
      Condition.signal q.not_full;  
      x)
```

q = []

# Blocking Queue

```
let def q =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is empty *)  
    while q.tail = q.head do  
      Condition.wait q.not_empty q.lock  
    done;  
  
    match q.items.(q.head mod q.capacity) with  
    | None -> assert false (* Should never happen *)  
    | Some x ->  
      q.items.(q.head mod q.capacity) <- None;  
      q.head <- q.head + 1;  
      (* Signal that queue is not full *)  
      Condition.signal q.not_full;  
      x)
```

def q

D1

q = []

q.lock

Critical Section

q.not\_empty

Waiting Area

# Blocking Queue

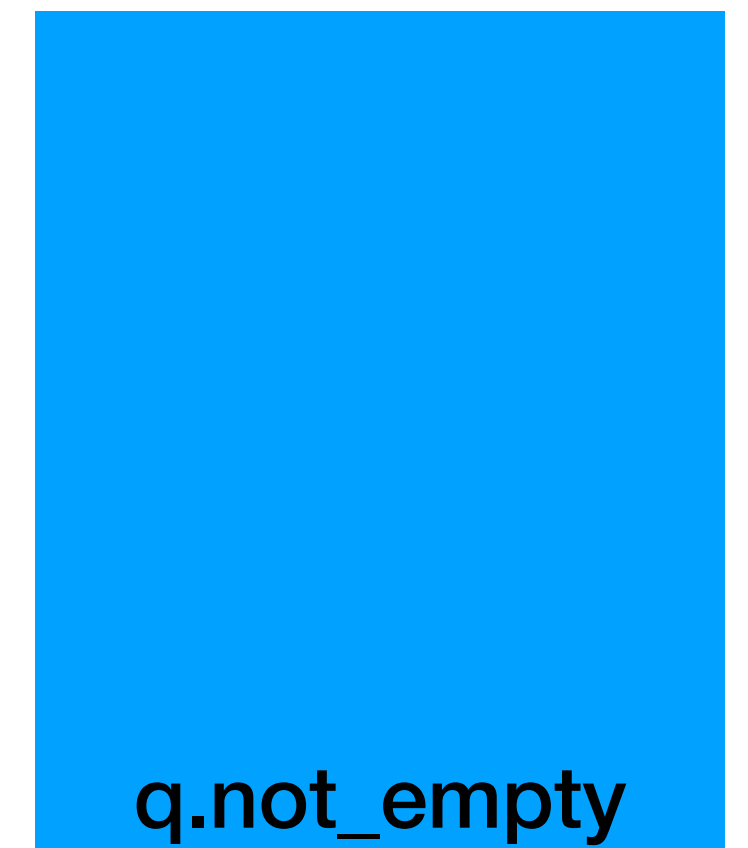
```
let def q =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is empty *)  
    while q.tail = q.head do  
      Condition.wait q.not_empty q.lock  
    done;  
  
    match q.items.(q.head mod q.capacity) with  
    | None -> assert false (* Should never happen *)  
    | Some x ->  
      q.items.(q.head mod q.capacity) <- None;  
      q.head <- q.head + 1;  
      (* Signal that queue is not full *)  
      Condition.signal q.not_full;  
      x)
```

q = []

def q



Critical Section



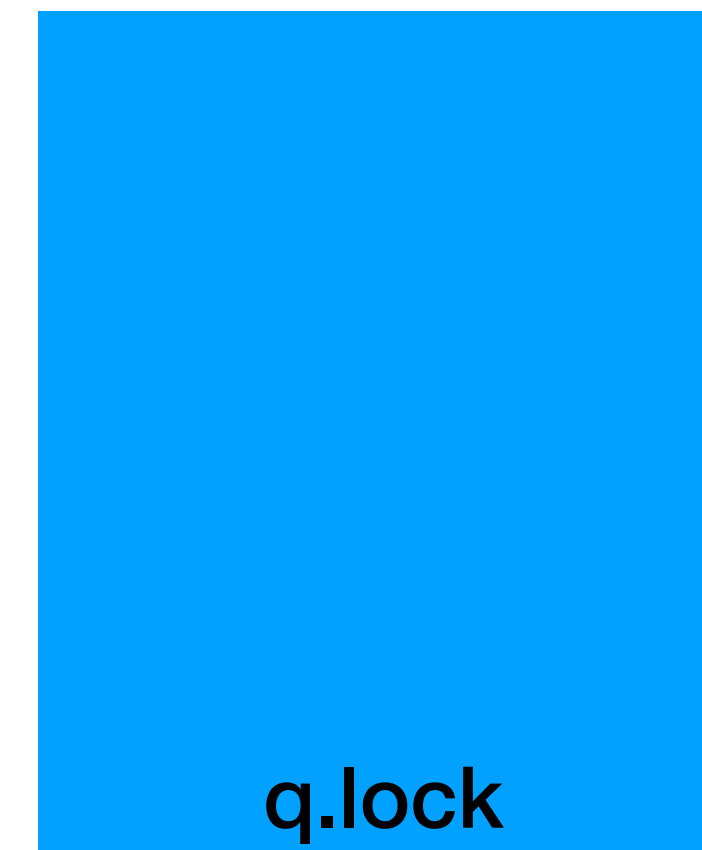
Waiting Area

# Blocking Queue

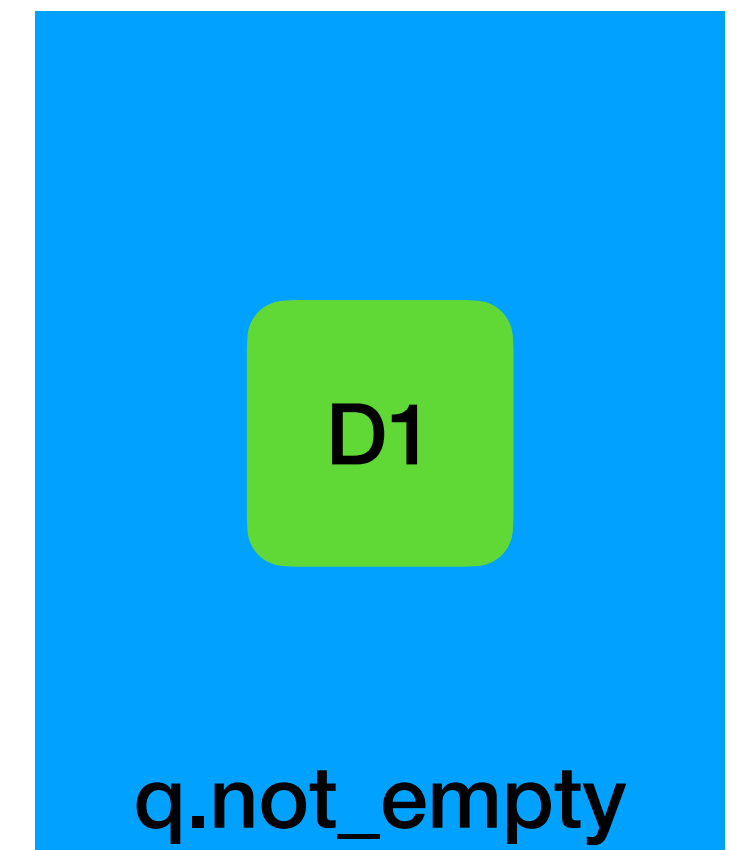
```
let def q =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is empty *)  
    while q.tail = q.head do  
      Condition.wait q.not_empty q.lock  
    done;  
  
    match q.items.(q.head mod q.capacity) with  
    | None -> assert false (* Should never happen *)  
    | Some x ->  
      q.items.(q.head mod q.capacity) <- None;  
      q.head <- q.head + 1;  
      (* Signal that queue is not full *)  
      Condition.signal q.not_full;  
      x)
```

q = []

def q



Critical Section



Waiting Area

# Blocking Queue

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
    (fun () ->  
      (* Wait while queue is full *)  
      while q.tail - q.head = q.capacity do  
        Condition.wait q.not_full q.lock  
      done;  
  
      (* Add element *)  
      q.items.(q.tail mod q.capacity) <- Some x;  
      q.tail <- q.tail + 1;  
  
      (* Signal that queue is not empty *)  
      Condition.signal q.not_empty)
```

$q = []$



Critical Section



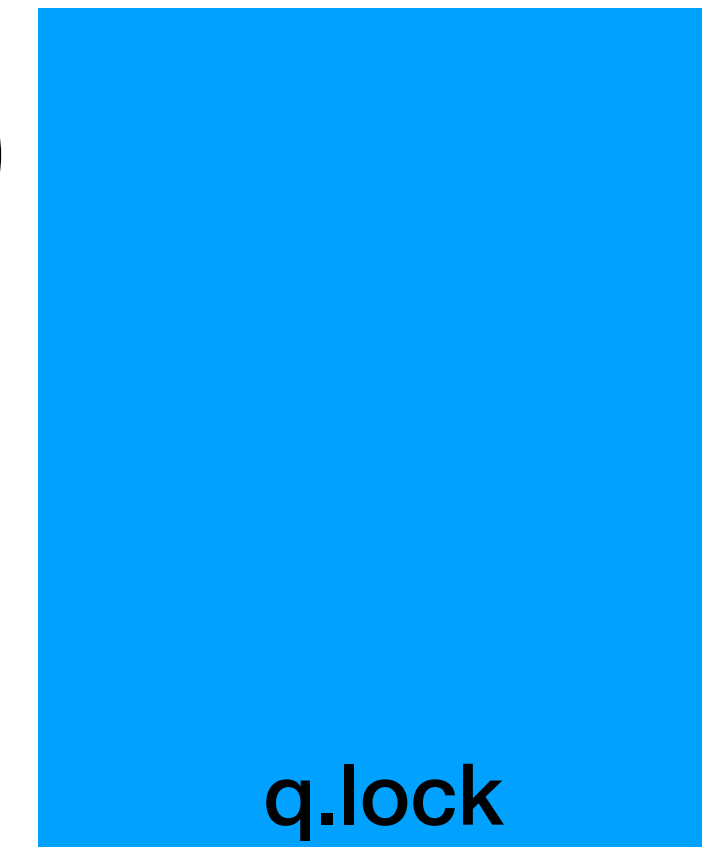
Waiting Area

# Blocking Queue

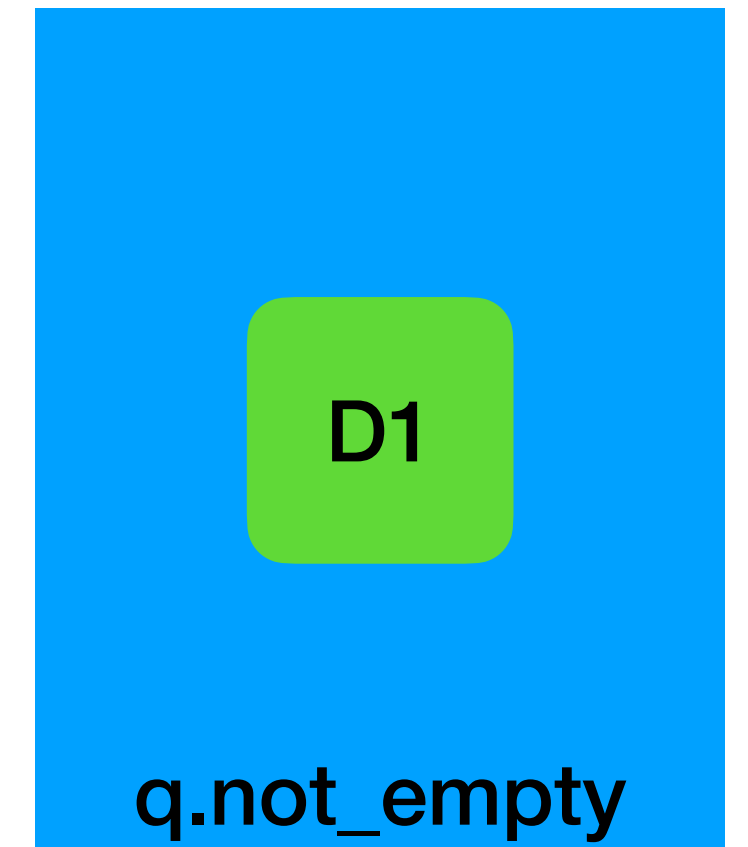
```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
    (fun () ->  
      (* Wait while queue is full *)  
      while q.tail - q.head = q.capacity do  
        Condition.wait q.not_full q.lock  
      done;  
  
      (* Add element *)  
      q.items.(q.tail mod q.capacity) <- Some x;  
      q.tail <- q.tail + 1;  
  
      (* Signal that queue is not empty *)  
      Condition.signal q.not_empty)
```

q = []

enq q 0



Critical Section



Waiting Area

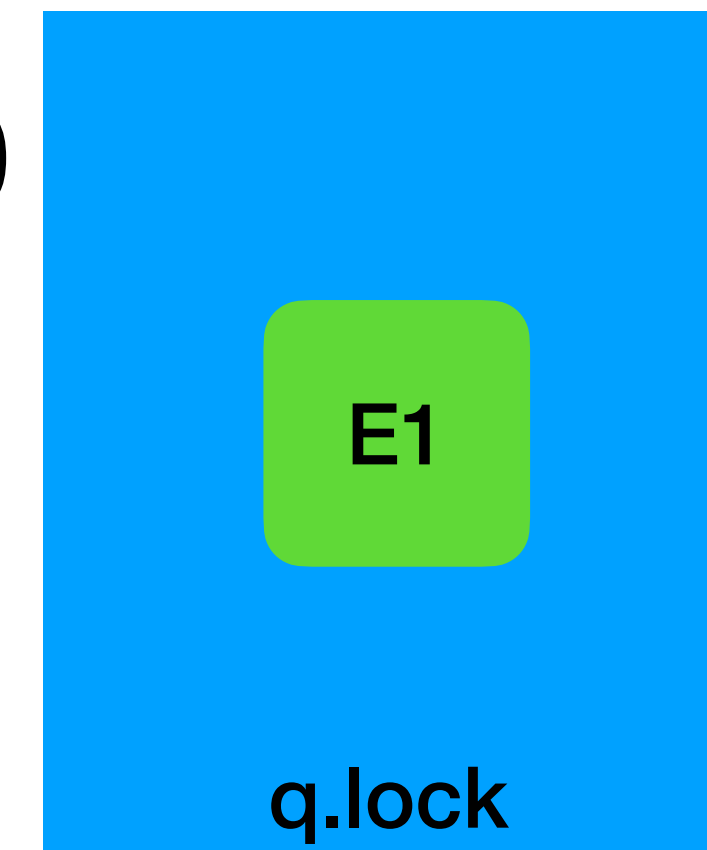


# Blocking Queue

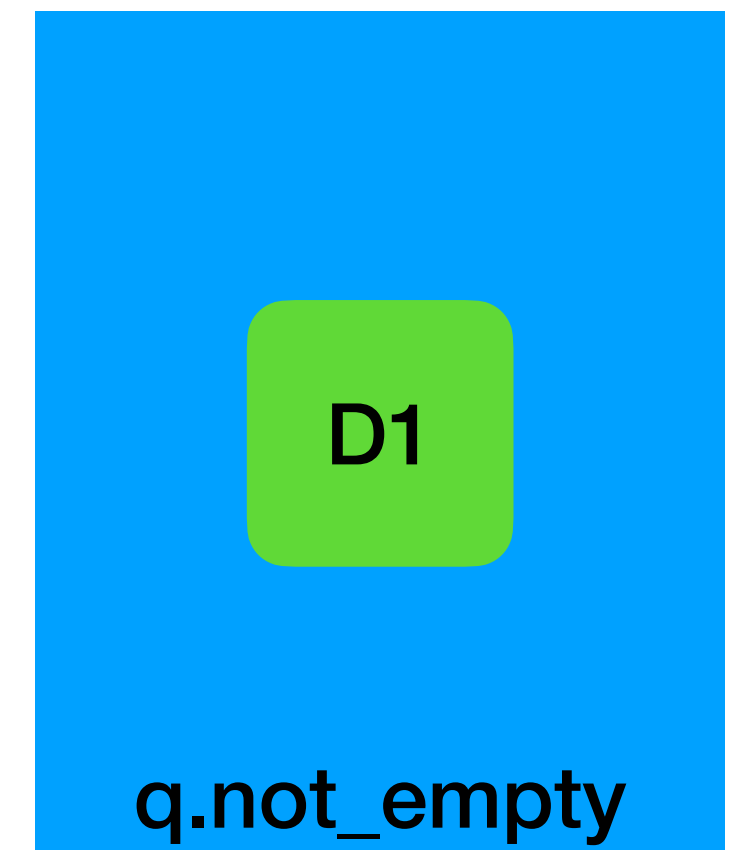
```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    Condition.signal q.not_empty)
```

q = []

enq q 0



Critical Section



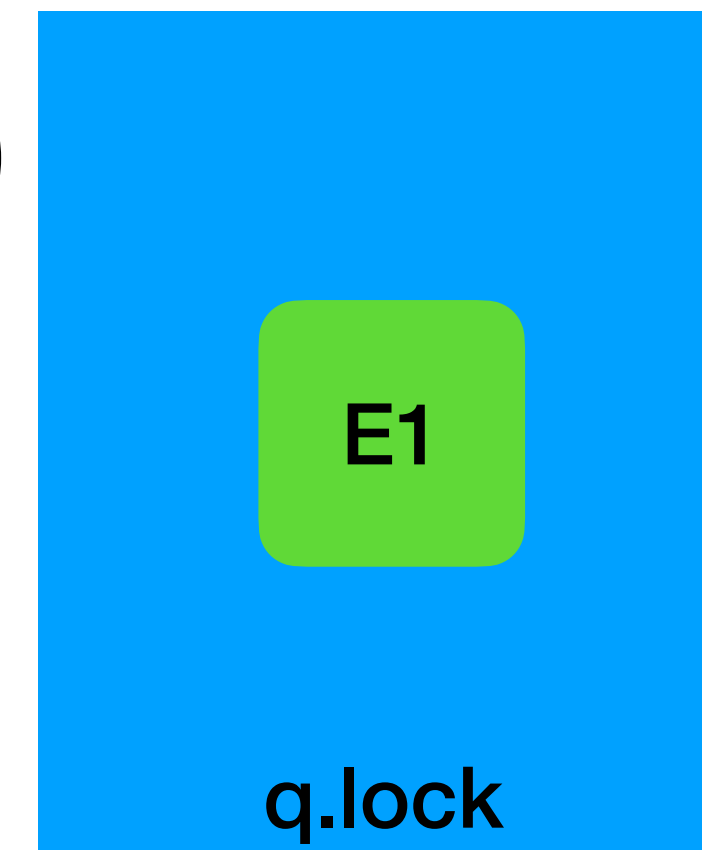
Waiting Area

# Blocking Queue

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
    (fun () ->  
      (* Wait while queue is full *)  
      while q.tail - q.head = q.capacity do  
        Condition.wait q.not_full q.lock  
      done;  
  
      (* Add element *)  
      q.items.(q.tail mod q.capacity) <- Some x;  
      q.tail <- q.tail + 1;  
  
      (* Signal that queue is not empty *)  
      Condition.signal q.not_empty)
```

$q = [0]$

enq q 0



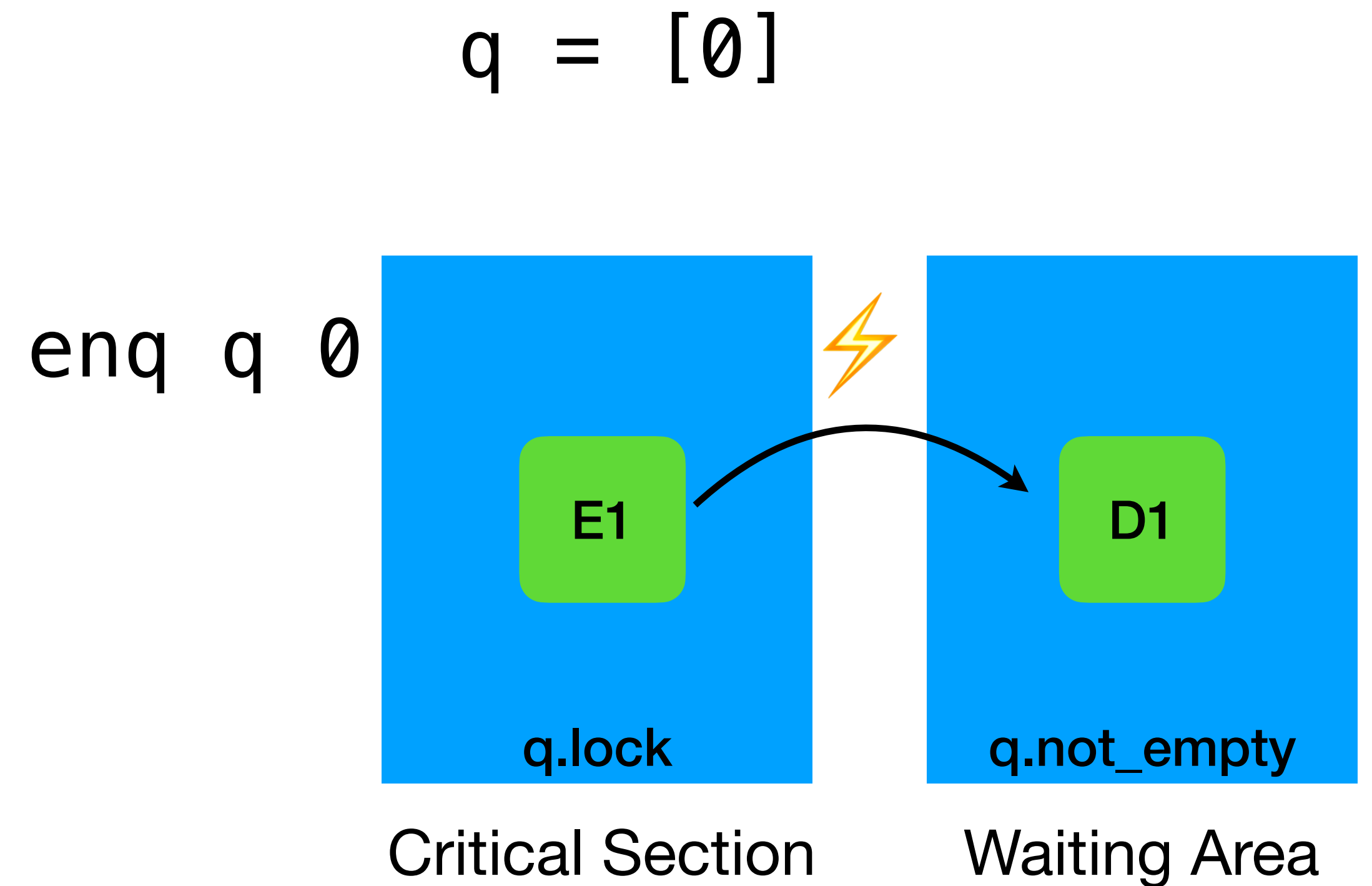
Critical Section



Waiting Area

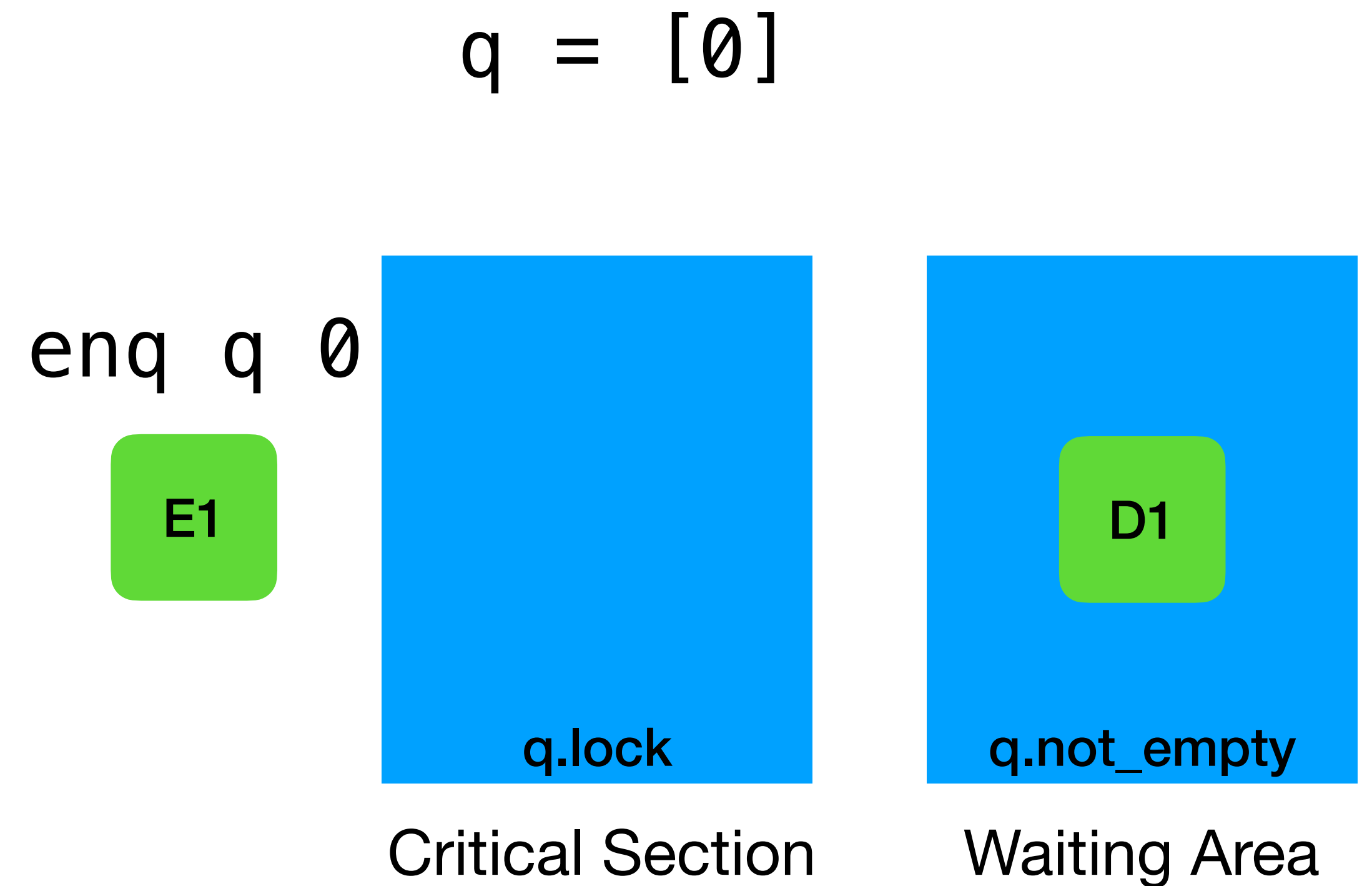
# Blocking Queue

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    Condition.signal q.not_empty)
```



# Blocking Queue

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
    (fun () ->  
      (* Wait while queue is full *)  
      while q.tail - q.head = q.capacity do  
        Condition.wait q.not_full q.lock  
      done;  
  
      (* Add element *)  
      q.items.(q.tail mod q.capacity) <- Some x;  
      q.tail <- q.tail + 1;  
  
      (* Signal that queue is not empty *)  
      Condition.signal q.not_empty)
```



# Subtleties — Recheck condition for spurious wakeups

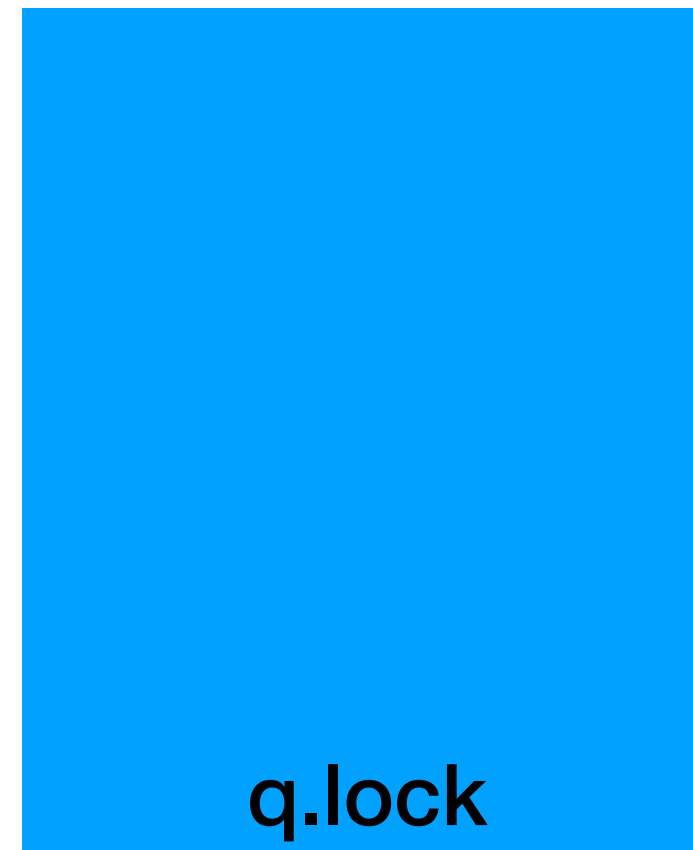
```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    Condition.signal q.not_empty)
```

```
let deq q =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is empty *)  
    while q.tail = q.head do  
      Condition.wait q.not_empty q.lock  
    done;  
  
    match q.items.(q.head mod q.capacity) with  
    | None -> assert false (* Should never happen *)  
    | Some x ->  
      q.items.(q.head mod q.capacity) <- None;  
      q.head <- q.head + 1;  
      (* Signal that queue is not full *)  
      Condition.signal q.not_full;  
      x)
```

*Spurious wakeups possible*

# Subtleties — Recheck condition for concurrency

`q = []`



Critical Section



Waiting Area

# Subtleties — Recheck condition for concurrency

$q = []$

enq q 0

E1

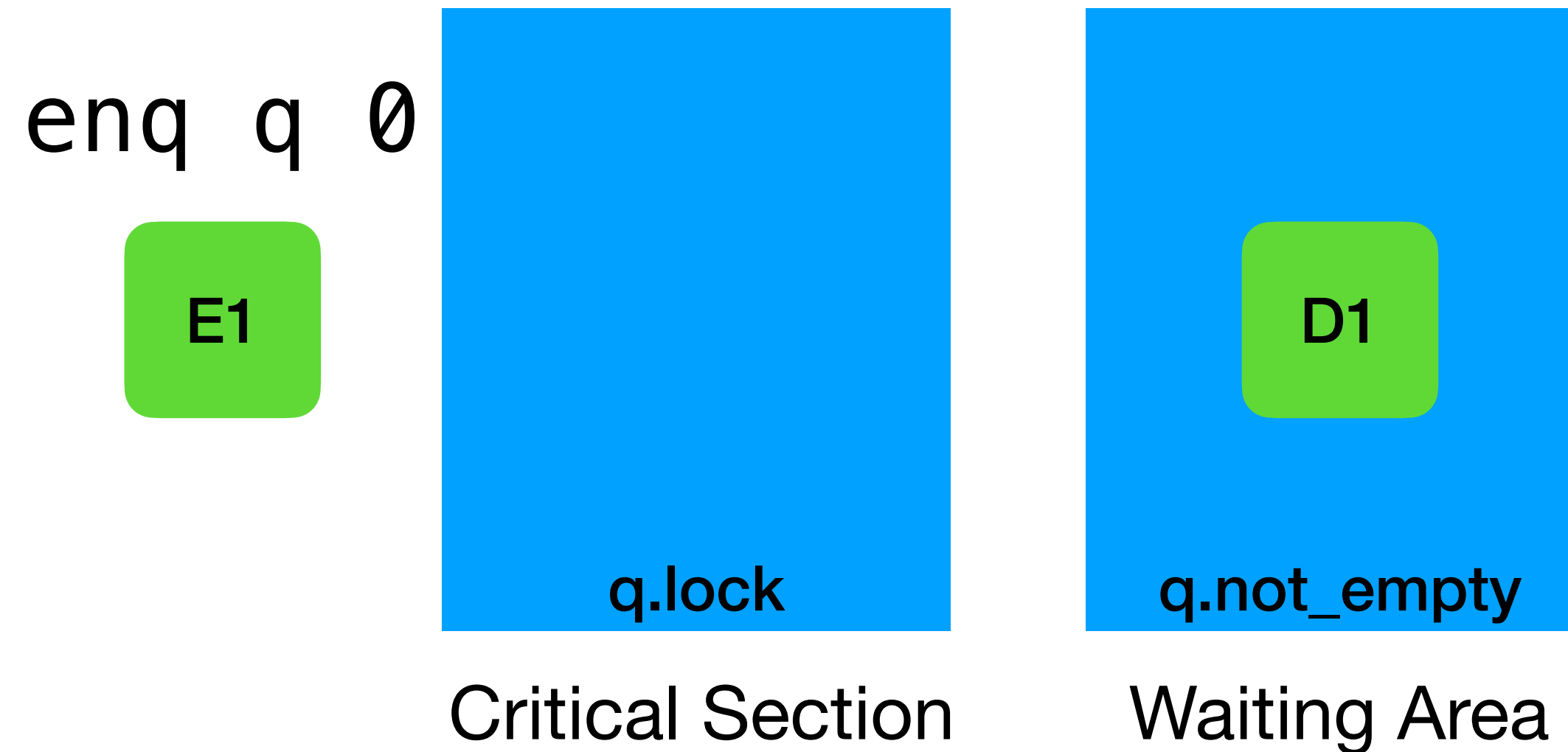
q.lock

Critical Section

D1

q.not\_empty

Waiting Area



# Subtleties — Recheck condition for concurrency

`q = []`

`enq q 0`



Critical Section



Waiting Area



# Subtleties — Recheck condition for concurrency

$q = [\emptyset]$

enq  $q$   $\emptyset$



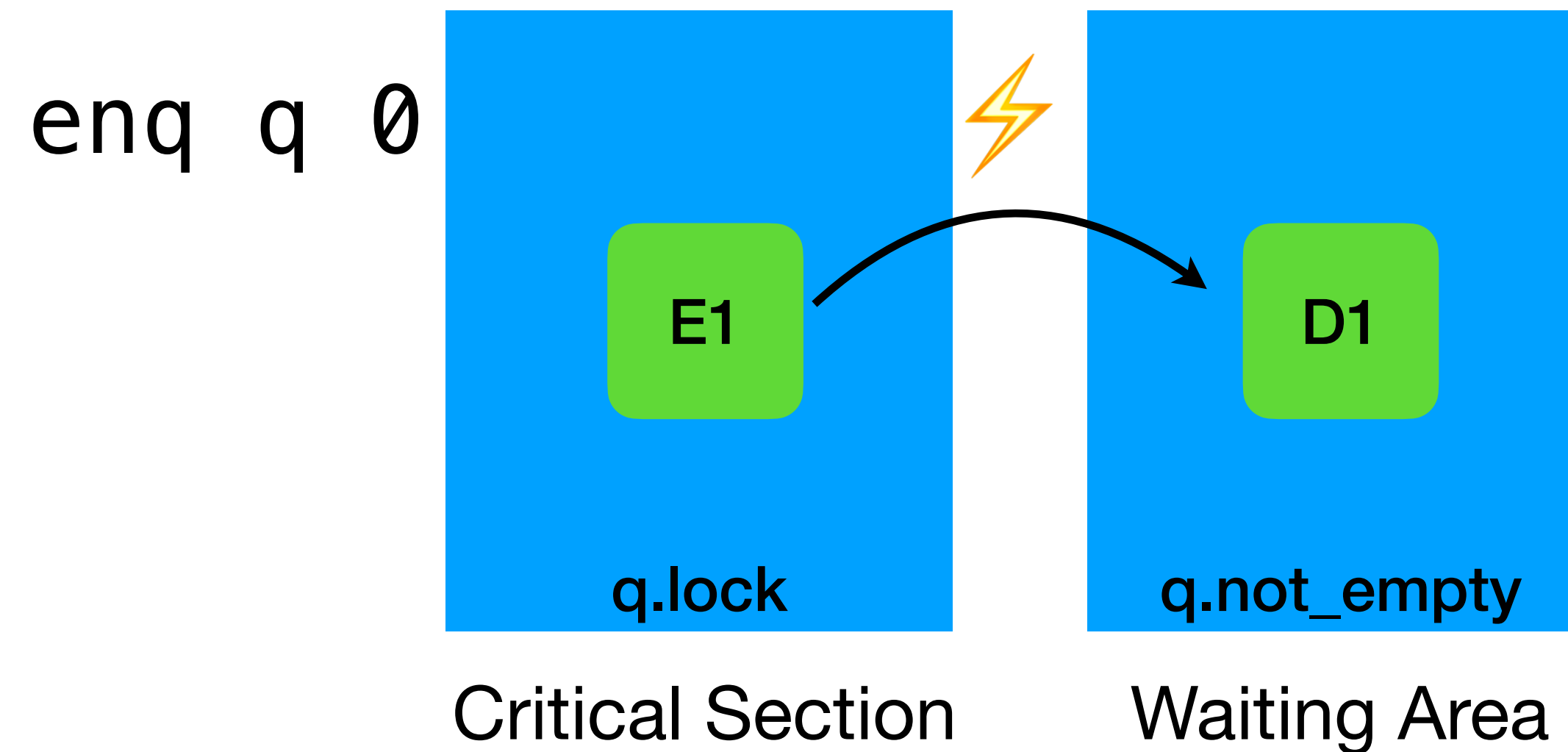
Critical Section



Waiting Area

# Subtleties — Recheck condition for concurrency

$q = [\emptyset]$



# Subtleties — Recheck condition for concurrency

$q = [\emptyset]$

enq q  $\emptyset$

E1

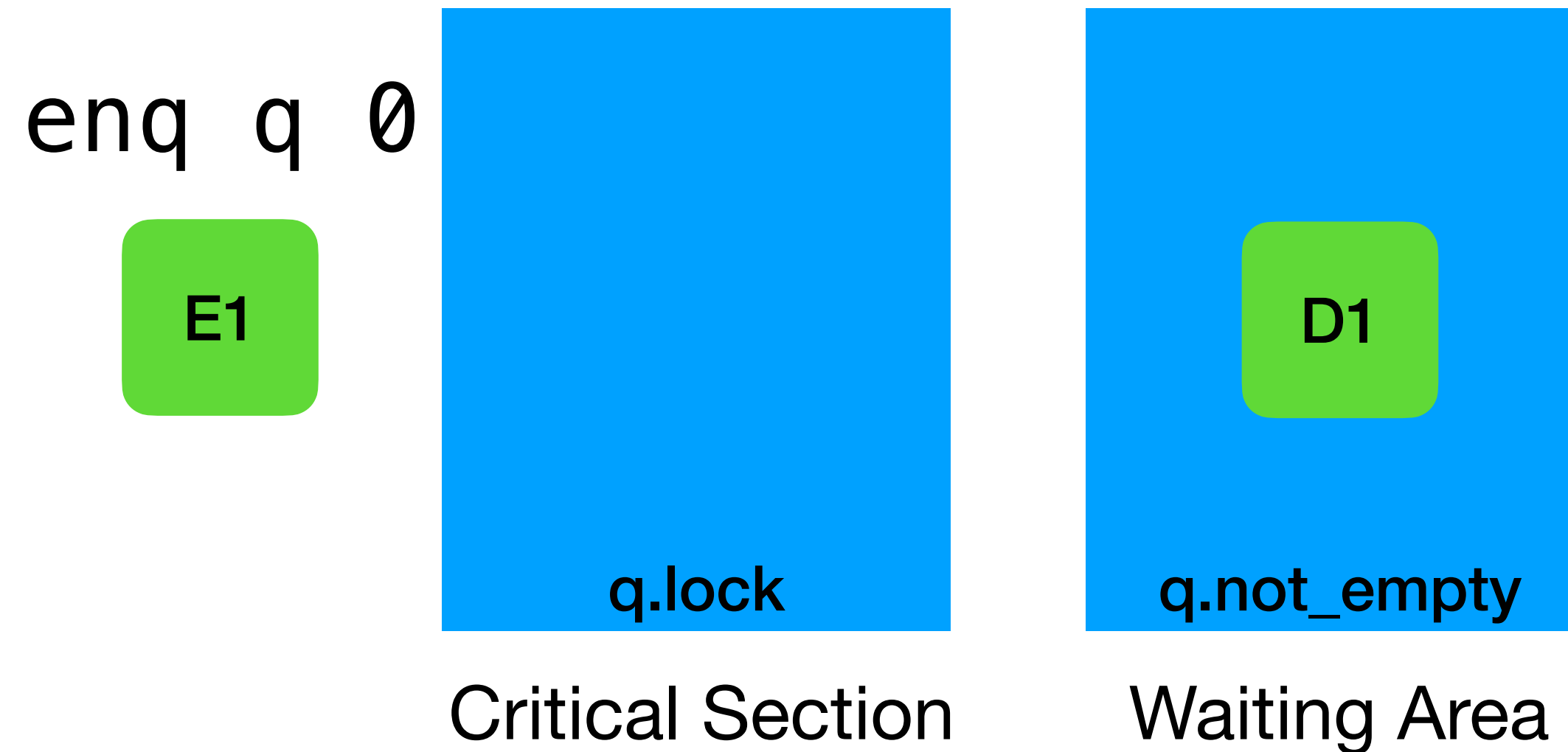
q.lock

Critical Section

D1

q.not\_empty

Waiting Area



# Subtleties — Recheck condition for concurrency

$q = [\emptyset]$

enq  $q$   $\emptyset$

E1

$q.lock$

Critical Section

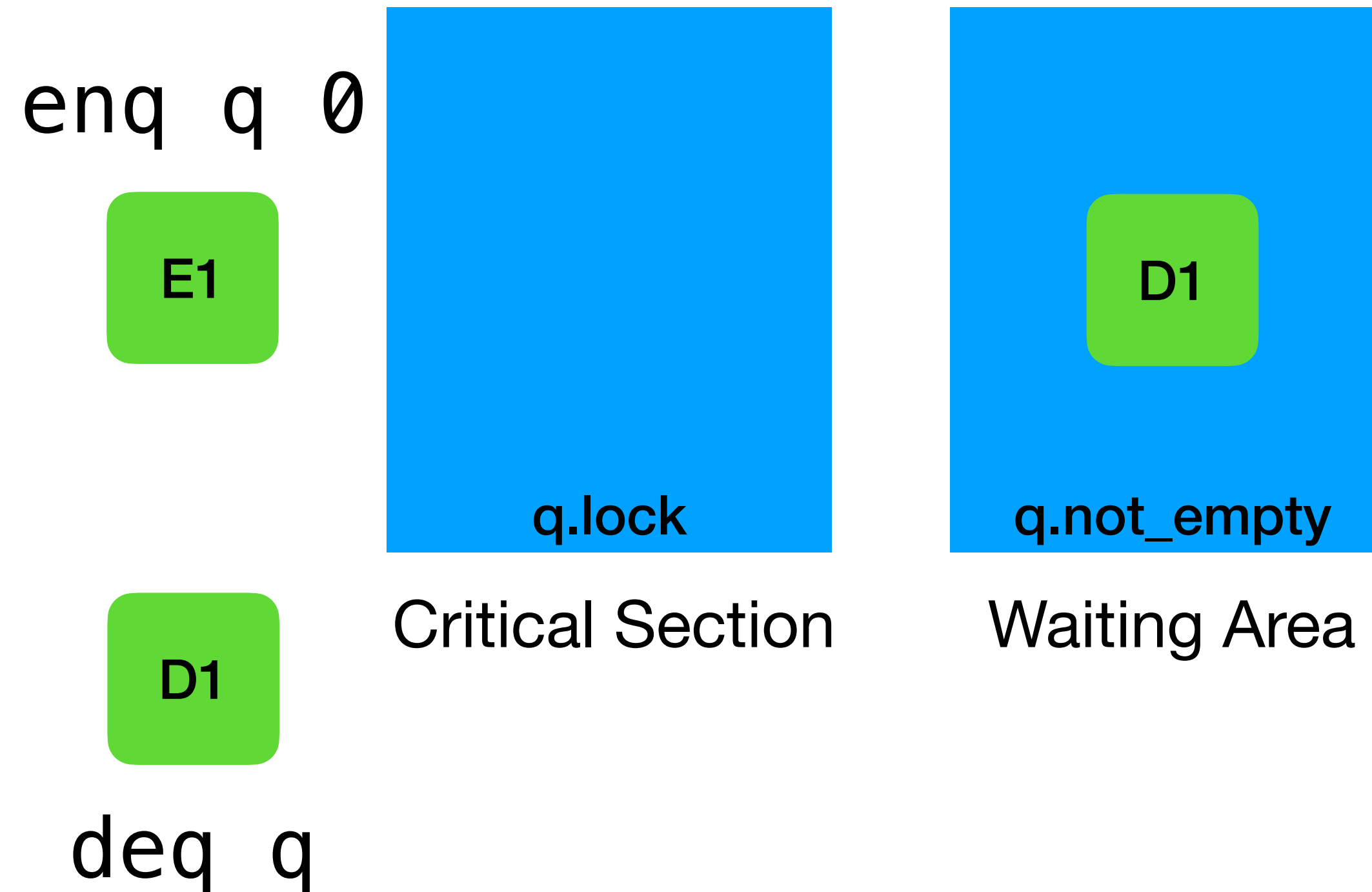
D1

deq  $q$

D1

$q.not\_empty$

Waiting Area



# Subtleties — Recheck condition for concurrency

$q = [\emptyset]$

enq  $q$   $\emptyset$



Critical Section



Waiting Area

deq  $q$

# Subtleties — Recheck condition for concurrency

$q = []$

enq  $q$   $\emptyset$



Critical Section



Waiting Area

deq  $q$

# Subtleties — Recheck condition for concurrency

$q = []$

enq  $q$   $\emptyset$

E1

$q.lock$

Critical Section

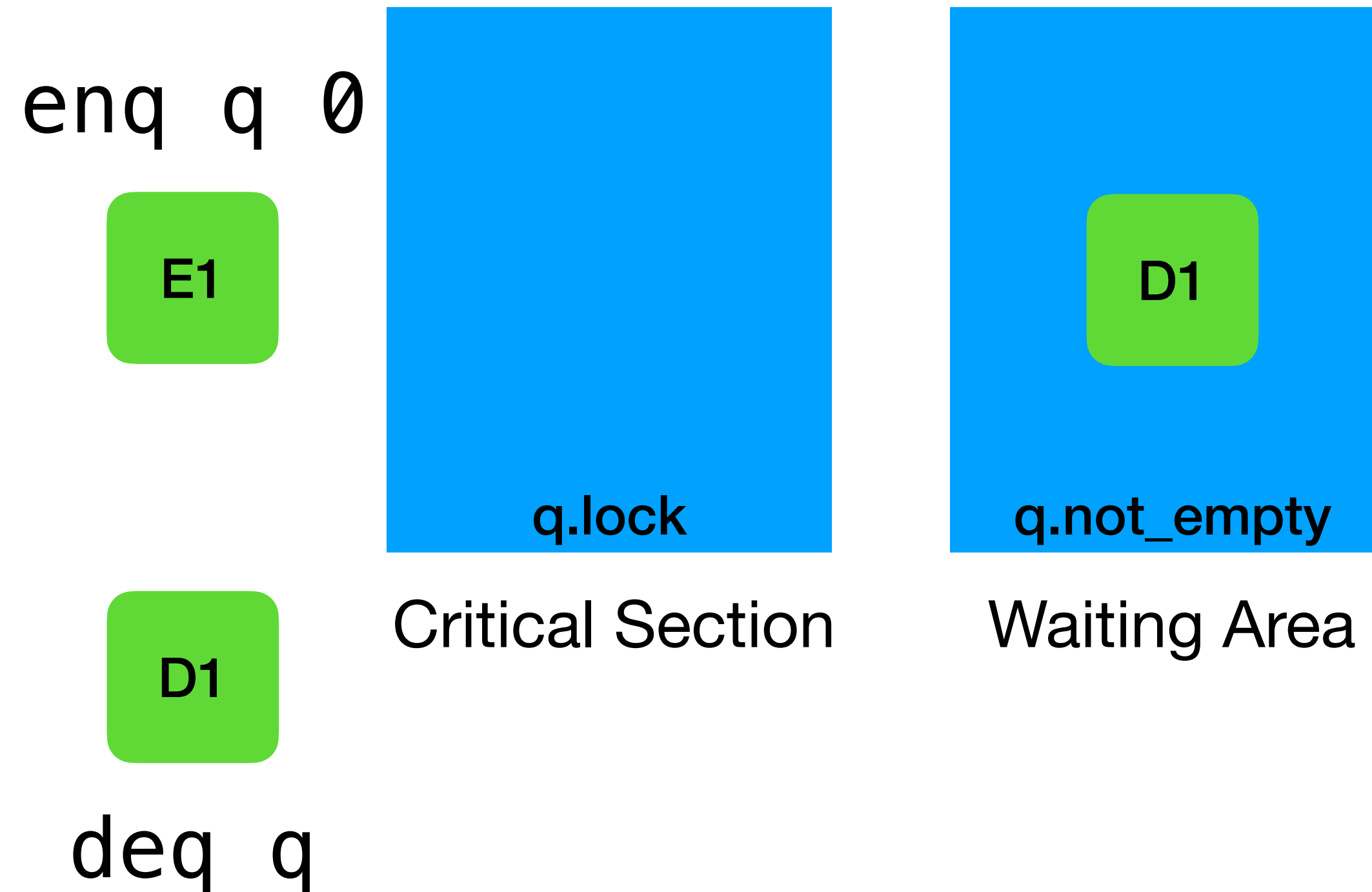
D1

deq  $q$

D1

$q.not\_empty$

Waiting Area



# Subtleties — Lost-wakeup Problem

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
    (fun () ->  
      (* Wait while queue is full *)  
      while q.tail - q.head = q.capacity do  
        Condition.wait q.not_full q.lock  
      done;  
  
      (* Add element *)  
      q.items.(q.tail mod q.capacity) <- Some x;  
      q.tail <- q.tail + 1;  
  
      (* Signal that queue is not empty *)  
      Condition.signal q.not_empty)
```

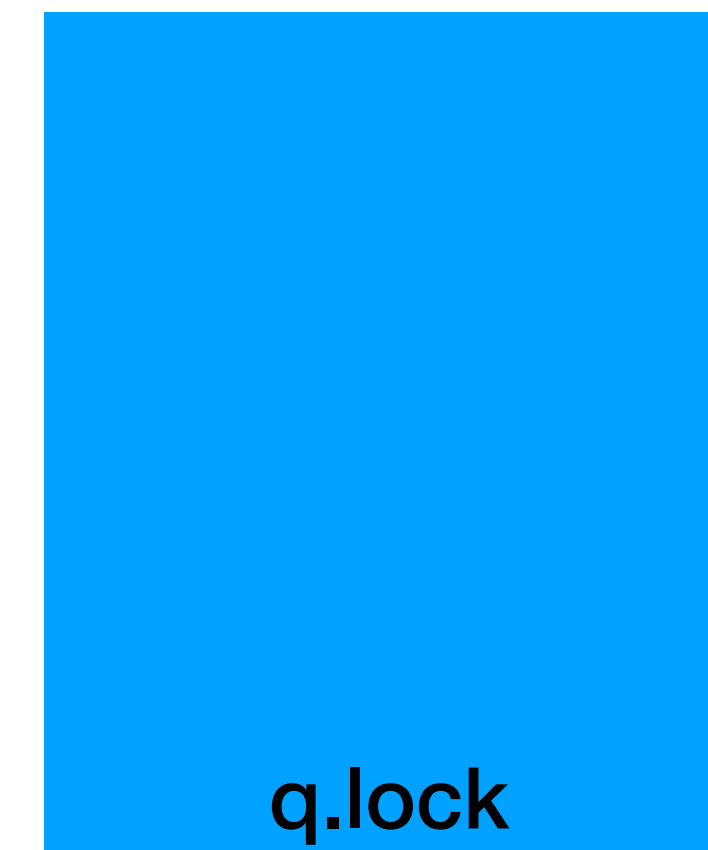
*Note that we signal all the time, not just when the queue transitions from empty to non-empty*



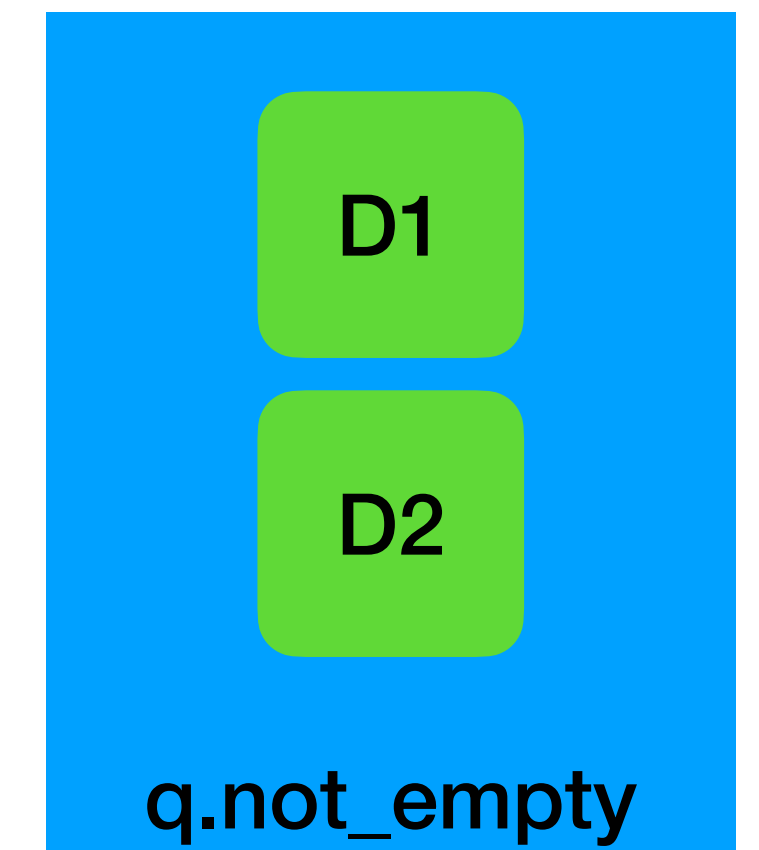
# Subtleties — Lost-wakeup Problem

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
    (* If queue is empty, we signal *)  
    let must_signal = q.tail = q.head in  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    if must_signal then  
      Condition.signal q.not_empty)
```

q = []



Critical Section



Waiting Area

# Subtleties — Lost-wakeup Problem

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
    (* If queue is empty, we signal *)  
    let must_signal = q.tail = q.head in  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    if must_signal then  
      Condition.signal q.not_empty)
```

q = []

enq q 0

E1

q.lock

Critical Section

D1

D2

q.not\_empty

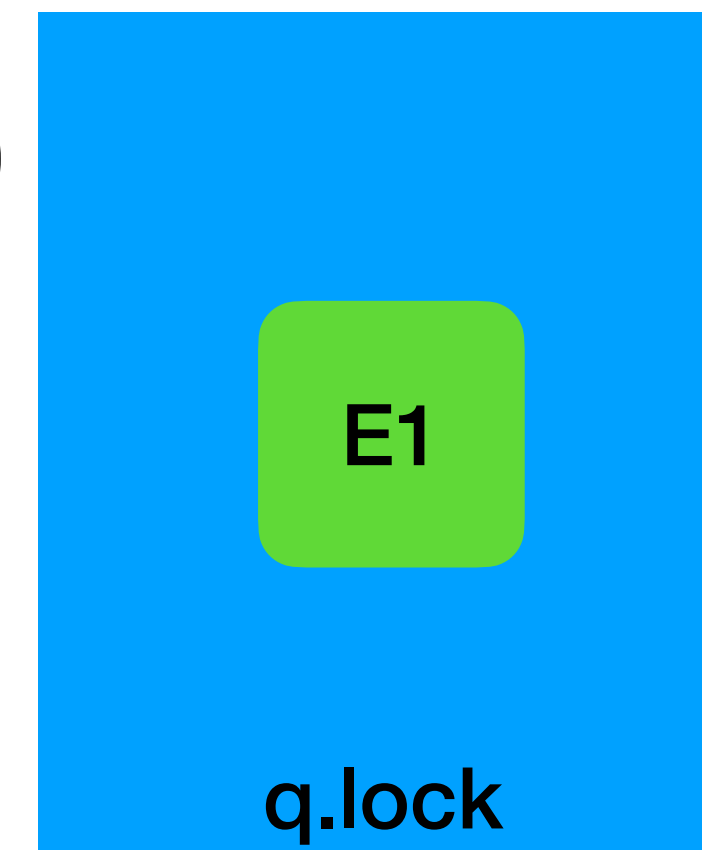
Waiting Area

# Subtleties — Lost-wakeup Problem

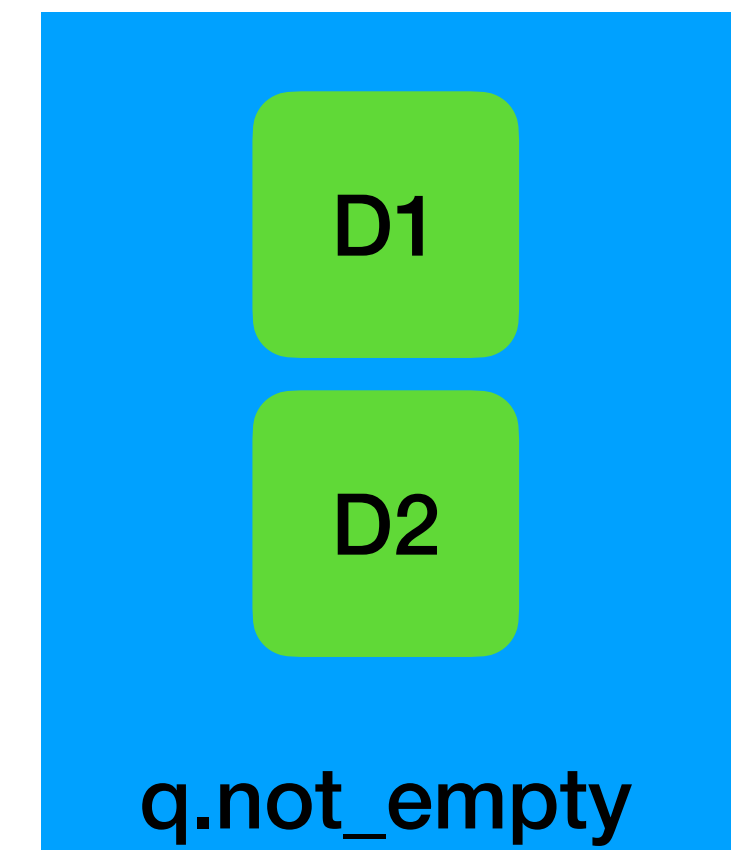
```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
    (* If queue is empty, we signal *)  
    let must_signal = q.tail = q.head in  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    if must_signal then  
      Condition.signal q.not_empty)
```

q = []

enq q 0



Critical Section



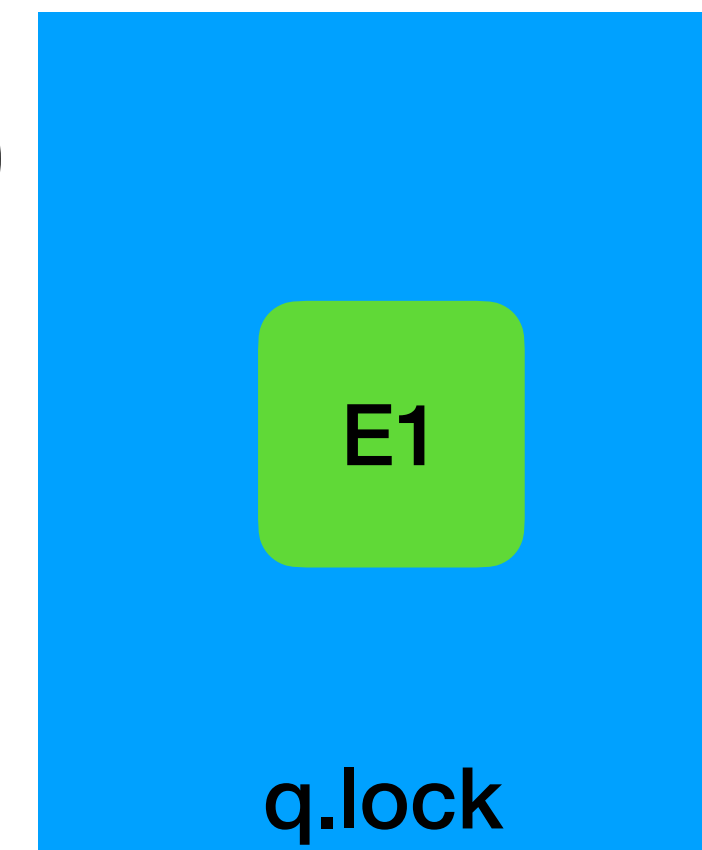
Waiting Area

# Subtleties — Lost-wakeup Problem

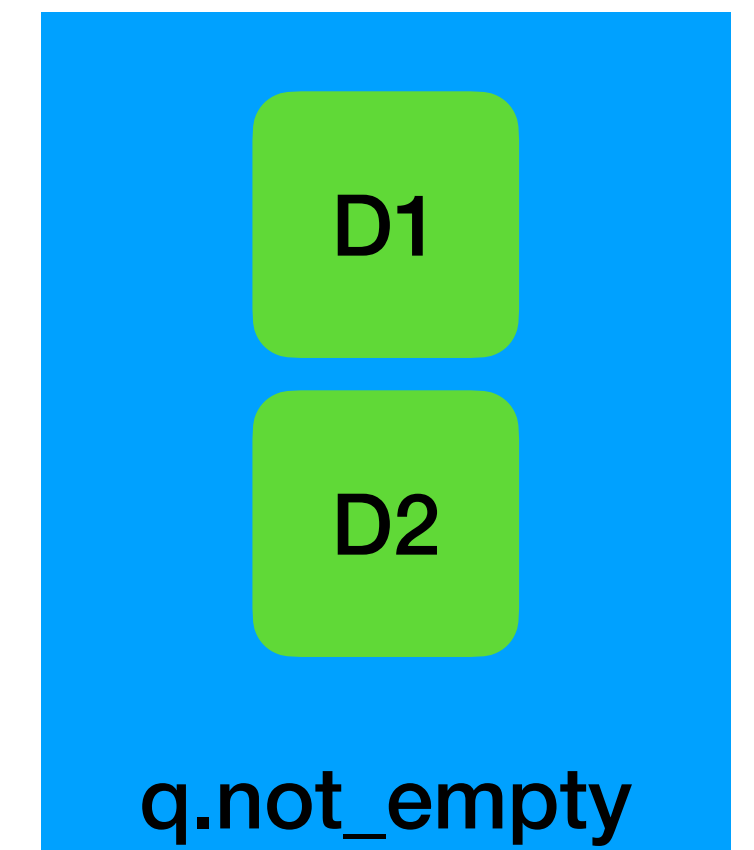
```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
    (* If queue is empty, we signal *)  
    let must_signal = q.tail = q.head in  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    if must_signal then  
      Condition.signal q.not_empty)
```

$q = [\emptyset]$

enq q 0



Critical Section

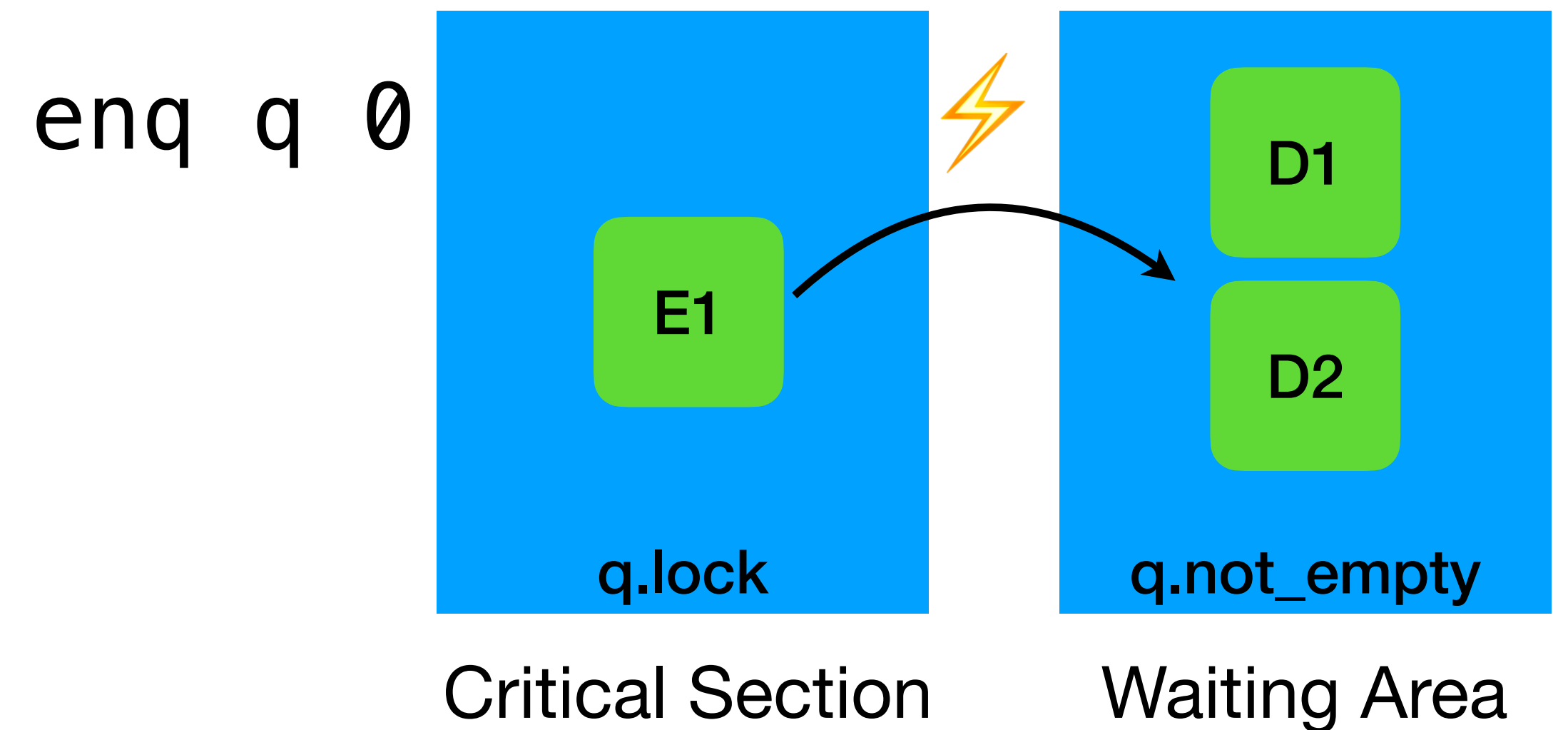


Waiting Area

# Subtleties — Lost-wakeup Problem

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
    (* If queue is empty, we signal *)  
    let must_signal = q.tail = q.head in  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    if must_signal then  
      Condition.signal q.not_empty)
```

$q = [\emptyset]$



# Subtleties — Lost-wakeup Problem

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
    (* If queue is empty, we signal *)  
    let must_signal = q.tail = q.head in  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    if must_signal then  
      Condition.signal q.not_empty)
```

q = [0]

enq q 0

E1

q.lock

Critical Section

D1

D2

q.not\_empty

Waiting Area

# Subtleties — Lost-wakeup Problem

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
    (* If queue is empty, we signal *)  
    let must_signal = q.tail = q.head in  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    if must_signal then  
      Condition.signal q.not_empty)
```

q = [0]

enq q 0

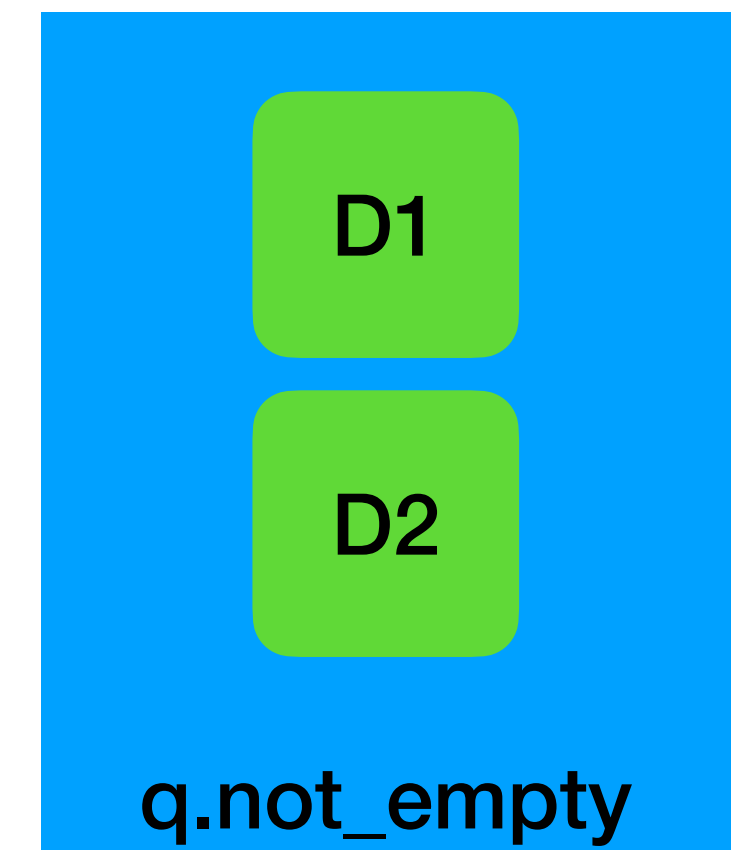
E1

E2

enq q 1



Critical Section



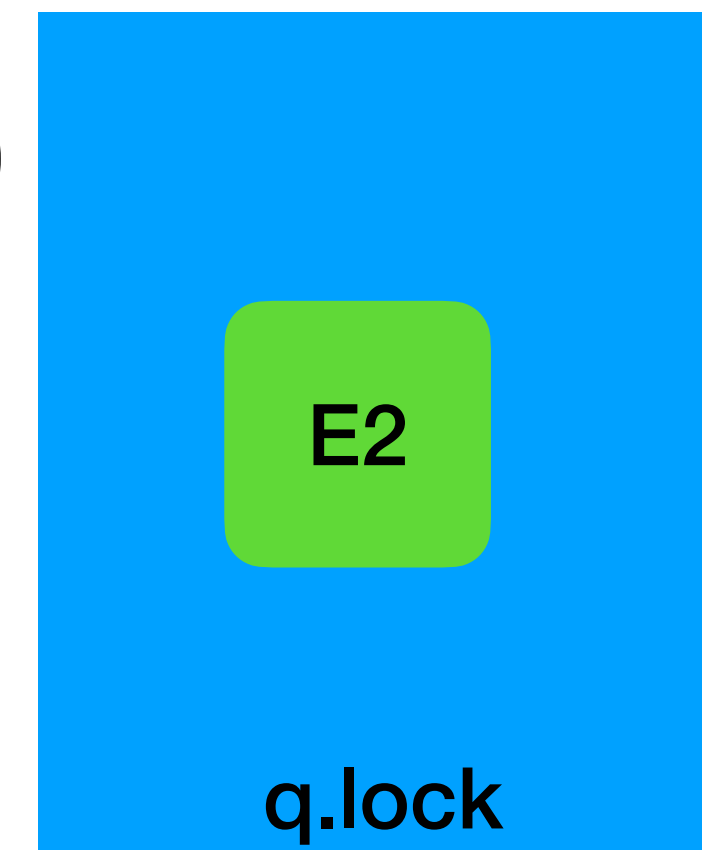
Waiting Area

# Subtleties — Lost-wakeup Problem

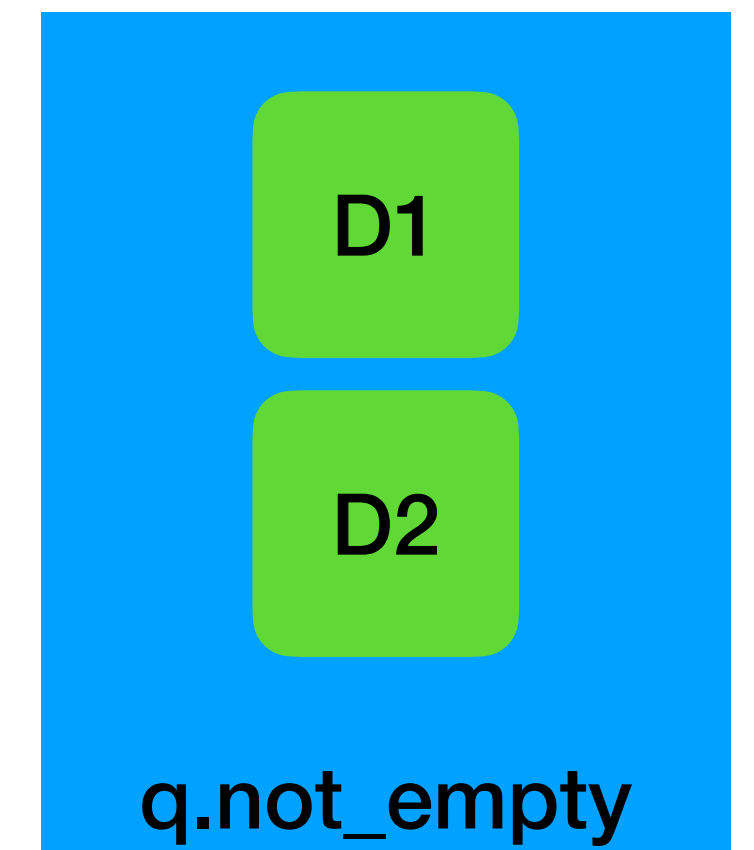
```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
    (fun () ->  
      (* Wait while queue is full *)  
      while q.tail - q.head = q.capacity do  
        Condition.wait q.not_full q.lock  
      done;  
      (* If queue is empty, we signal *)  
      let must_signal = q.tail = q.head in  
  
      (* Add element *)  
      q.items.(q.tail mod q.capacity) <- Some x;  
      q.tail <- q.tail + 1;  
  
      (* Signal that queue is not empty *)  
      if must_signal then  
        Condition.signal q.not_empty)
```

q = [0]

enq q 0



Critical Section



Waiting Area

enq q 1



# Subtleties — Lost-wakeup Problem

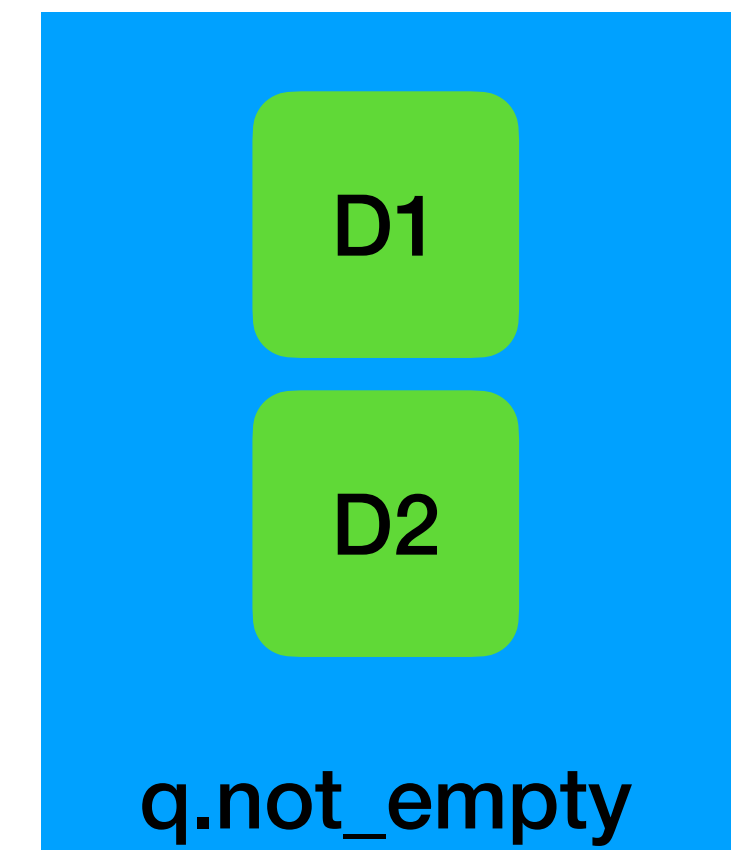
```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
    (* If queue is empty, we signal *)  
    let must_signal = q.tail = q.head in  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    if must_signal then  
      Condition.signal q.not_empty)
```

$q = [0; 1]$

enq q 0



Critical Section



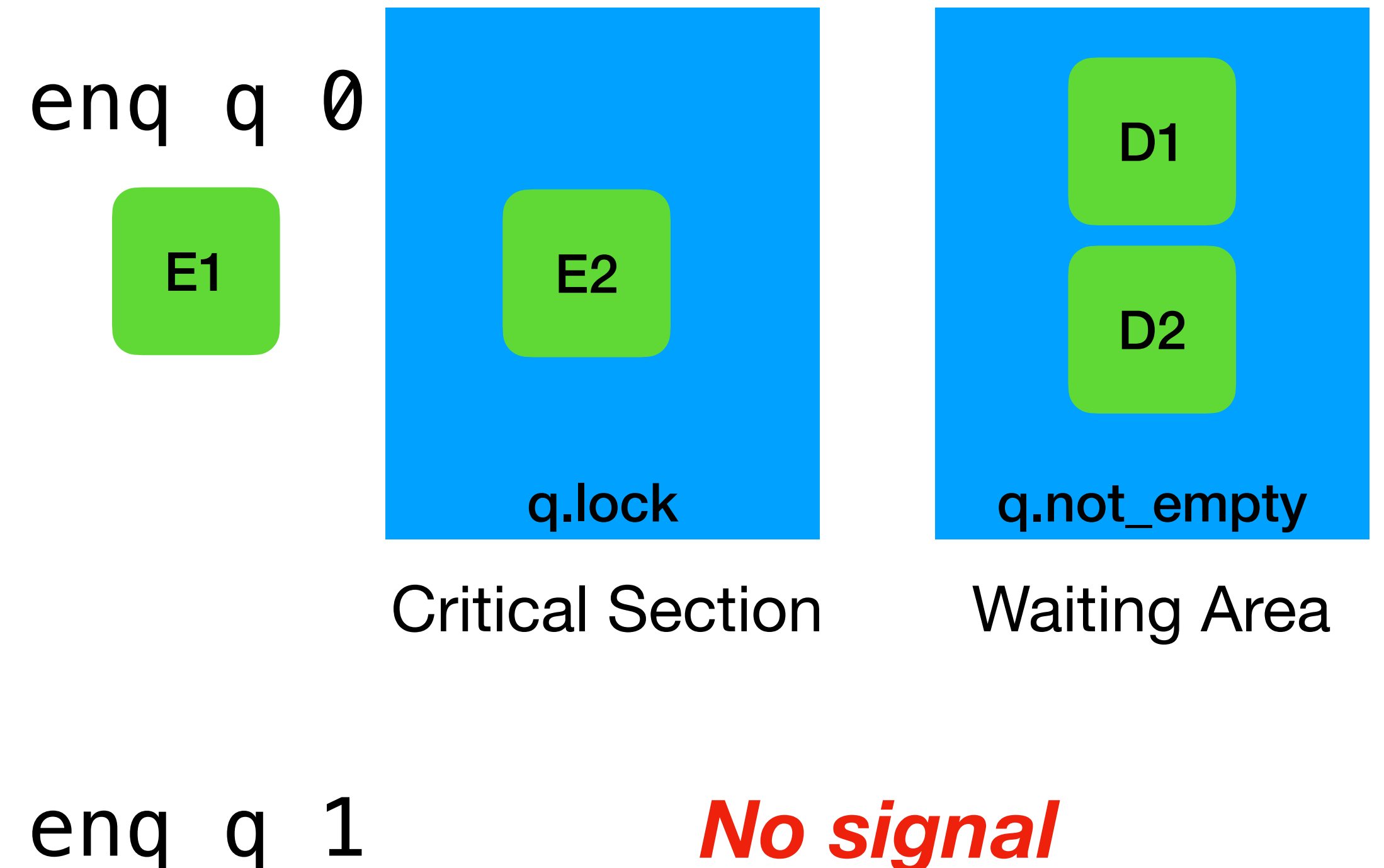
Waiting Area

enq q 1

# Subtleties — Lost-wakeup Problem

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
    (* If queue is empty, we signal *)  
    let must_signal = q.tail = q.head in  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    if must_signal then  
      Condition.signal q.not_empty)
```

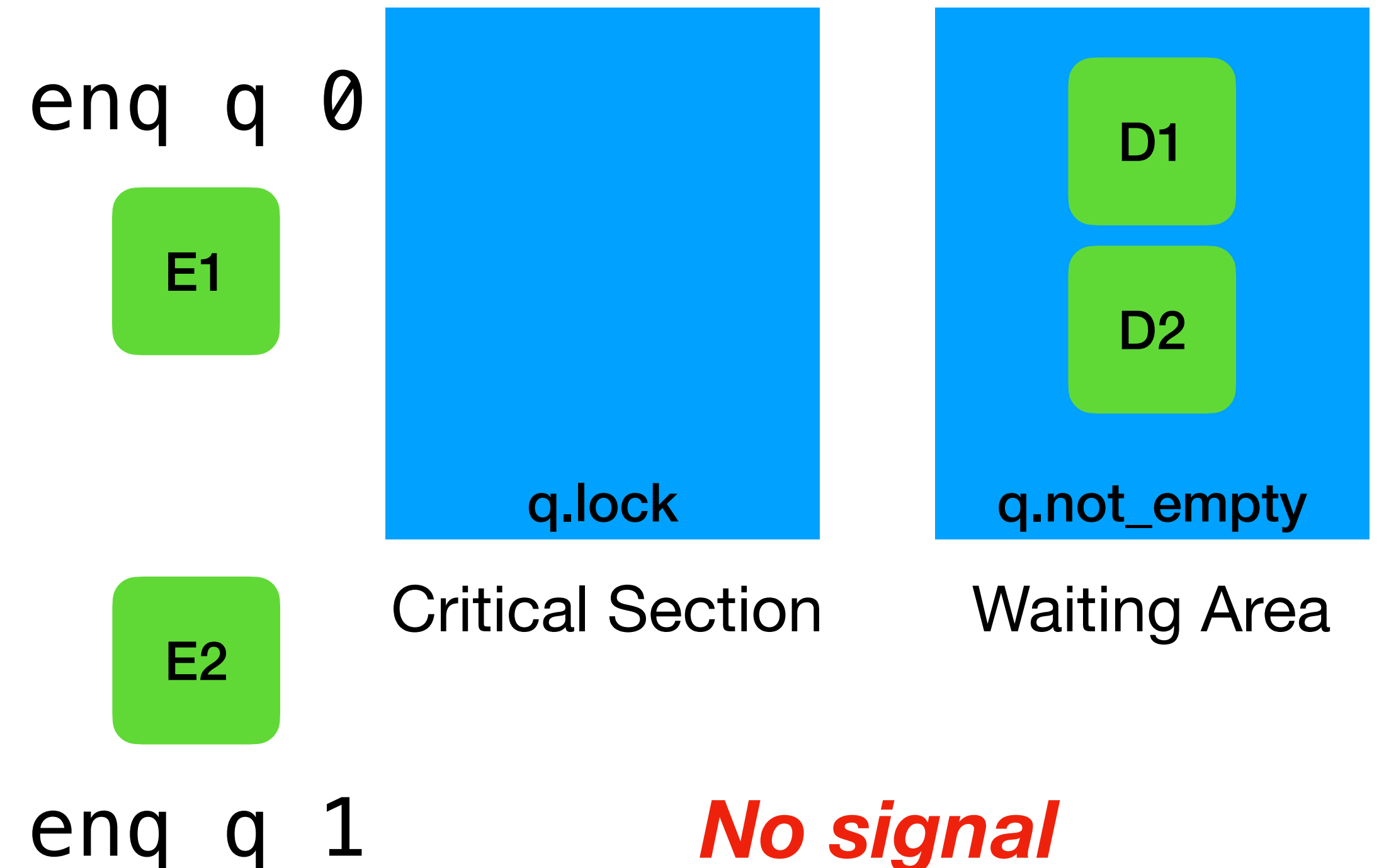
$q = [0; 1]$



# Subtleties — Lost-wakeup Problem

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
    (* If queue is empty, we signal *)  
    let must_signal = q.tail = q.head in  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    if must_signal then  
      Condition.signal q.not_empty)
```

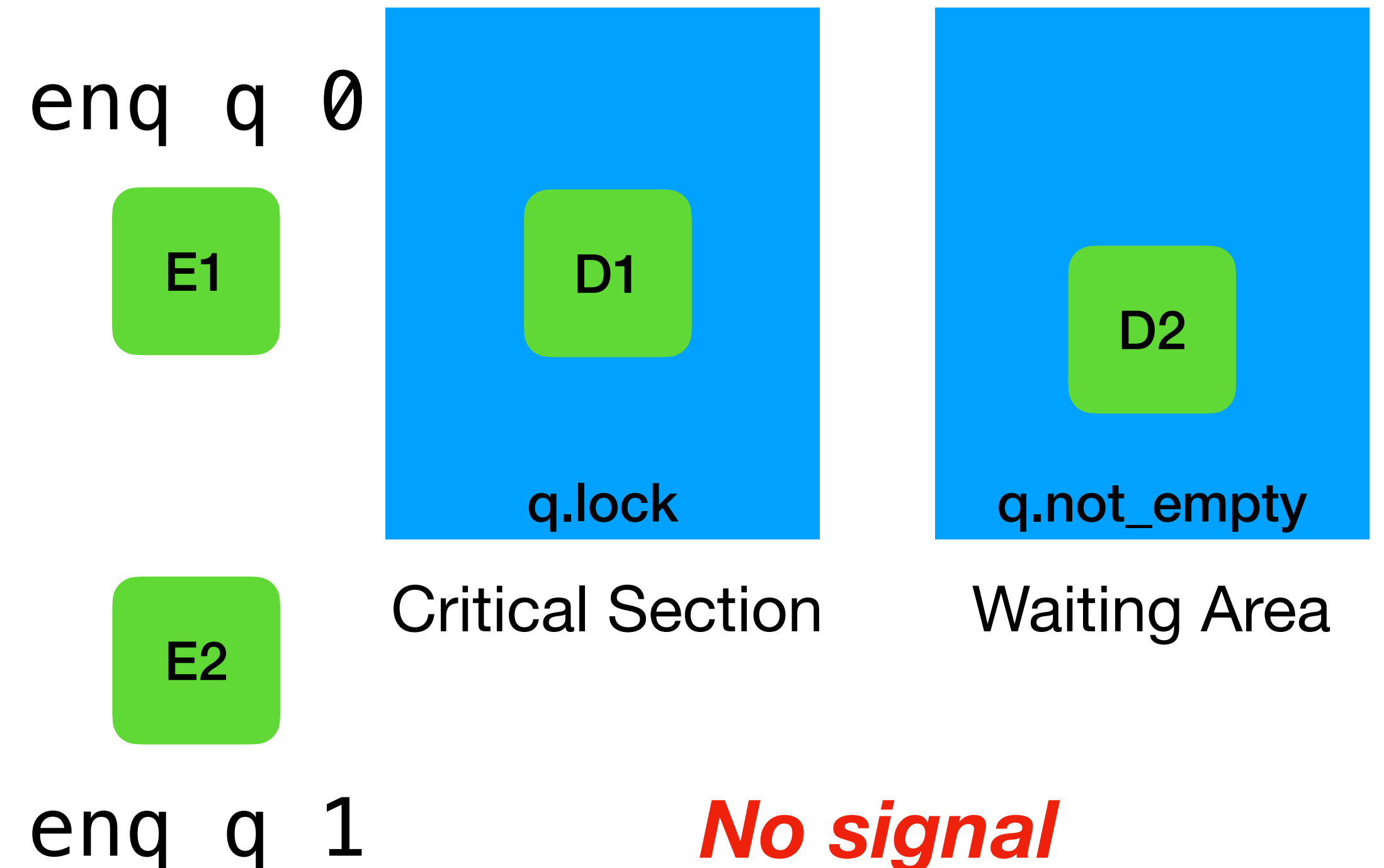
$q = [0; 1]$



# Subtleties — Lost-wakeup Problem

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
    (* If queue is empty, we signal *)  
    let must_signal = q.tail = q.head in  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    if must_signal then  
      Condition.signal q.not_empty)
```

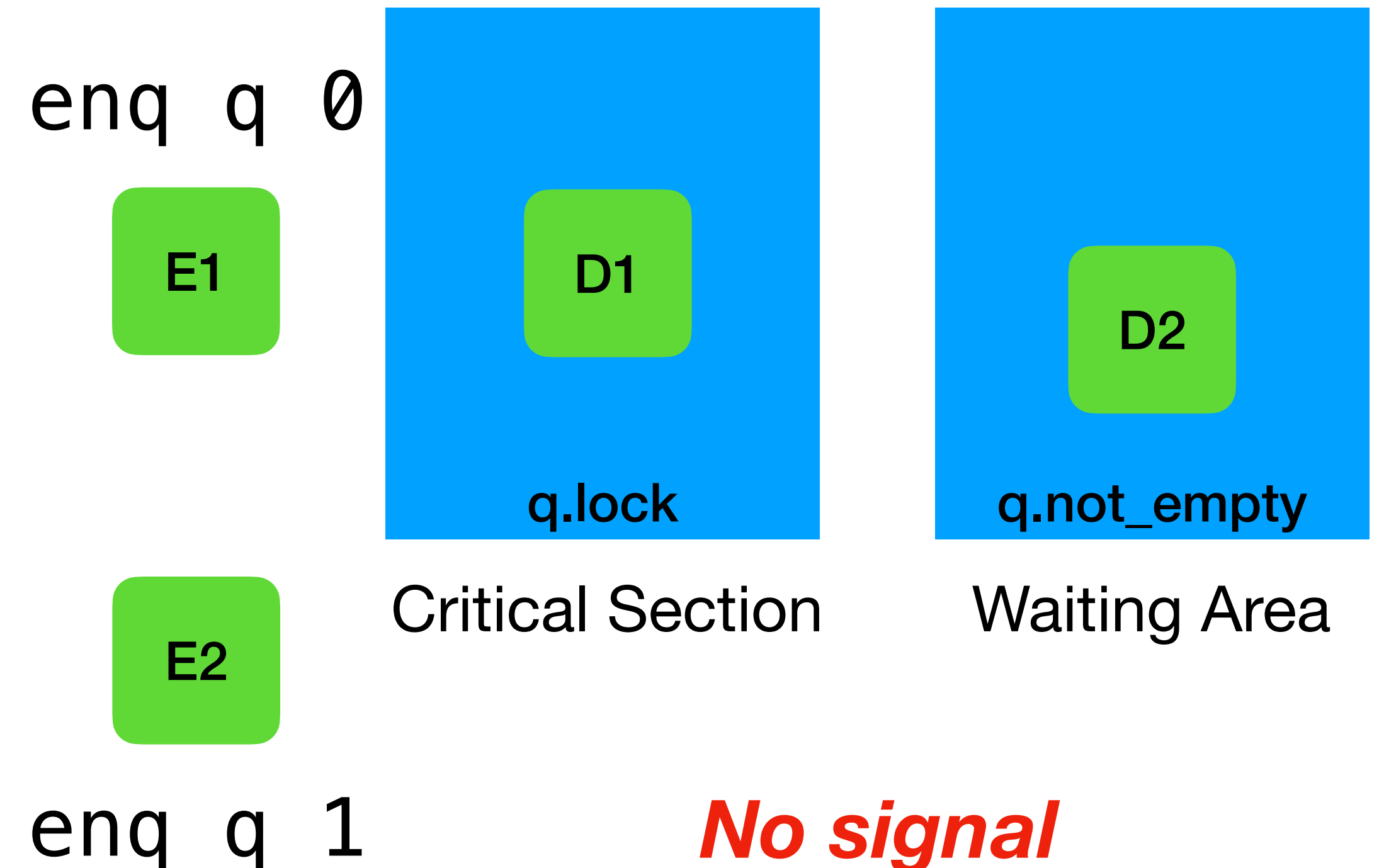
$q = [0; 1]$



# Subtleties — Lost-wakeup Problem

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
    (* If queue is empty, we signal *)  
    let must_signal = q.tail = q.head in  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    if must_signal then  
      Condition.signal q.not_empty)
```

q = [1]



# Subtleties — Lost-wakeup Problem

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
    (* If queue is empty, we signal *)  
    let must_signal = q.tail = q.head in  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    if must_signal then  
      Condition.signal q.not_empty)
```

q = [1]

enq q 0

E1

D1

E2

enq q 1



Critical Section



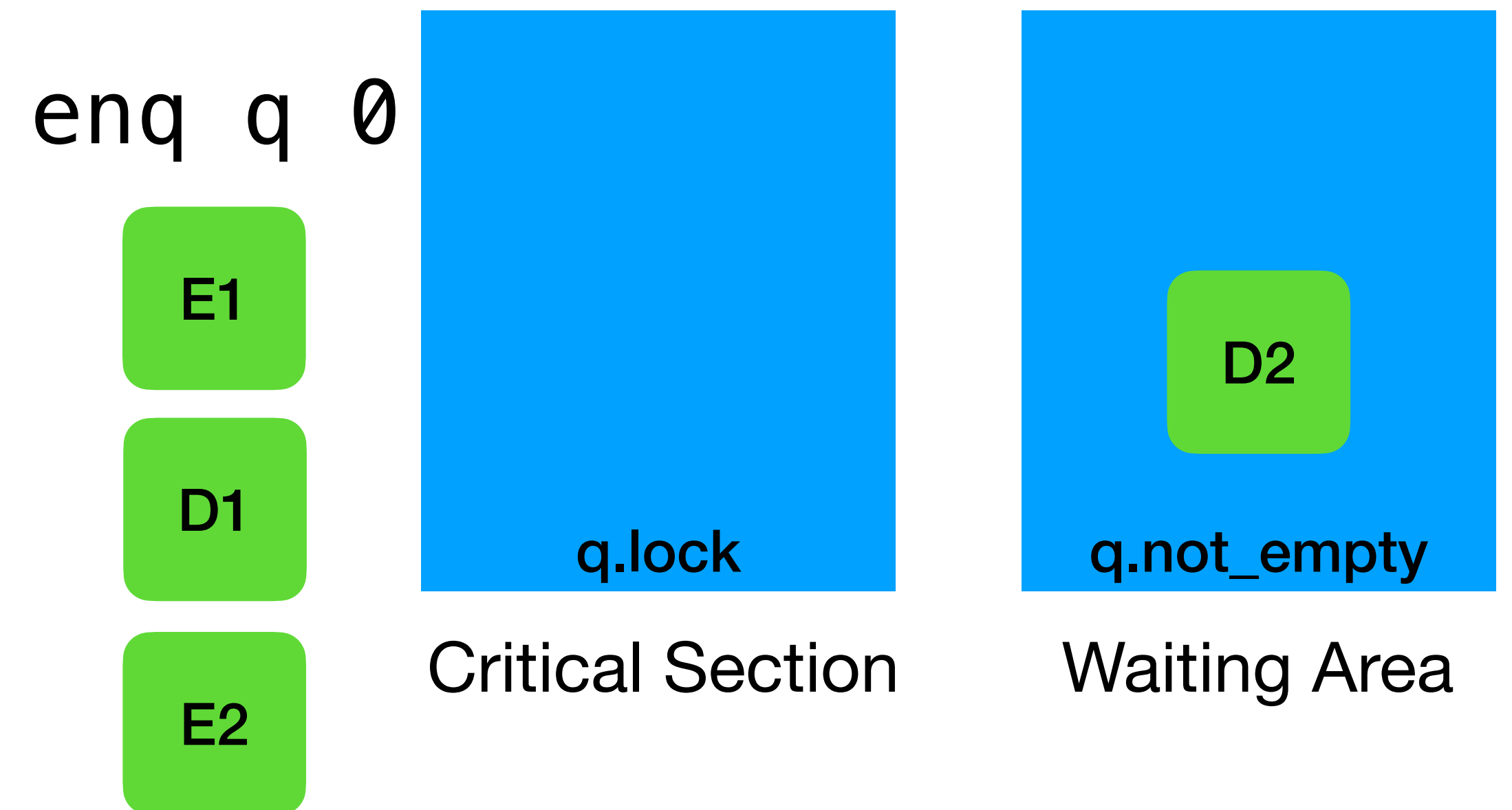
Waiting Area

**No signal**

# Subtleties — Lost-wakeup Problem

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
    (* If queue is empty, we signal *)  
    let must_signal = q.tail = q.head in  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    if must_signal then  
      Condition.signal q.not_empty)
```

q = [1]



enq q 1

**No signal**

***D2 still waiting even though the queue is non empty***

# Avoiding Lost wakeup problem

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    Condition.signal q.not_empty)
```

*Signal all the time*



# Avoiding Lost wakeup problem

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    Condition.signal q.not_empty)
```

*Signal all the time*

```
let enq q x =  
  Mutex.lock q.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock q.lock)  
  (fun () ->  
    (* Wait while queue is full *)  
    while q.tail - q.head = q.capacity do  
      Condition.wait q.not_full q.lock  
    done;  
  
    (* If queue is empty, we signal *)  
    let must_signal = q.tail = q.head in  
  
    (* Add element *)  
    q.items.(q.tail mod q.capacity) <- Some x;  
    q.tail <- q.tail + 1;  
  
    (* Signal that queue is not empty *)  
    if must_signal then  
      Condition.broadcast q.not_empty)
```

*Signal all the waiters when transitioning*

# Readers–Writers Locks

- Common pattern — *Read* shared resource frequently, but *modify* rarely
- Read-write Lock Invariant
  - Can't get write lock when read or write lock is held
  - Can't get read lock when write lock is head
- *Multiple readers can concurrently hold the lock!*
- **Note:** 1st edition of the AMP book has the wrong algorithm for read-write locks
  - 2nd edition has the right algorithm

# Simple Read-write Lock

```
type t
```

```
val create : unit -> t
```

```
val read_lock : t -> unit
```

```
val read_unlock : t -> unit
```

```
val write_lock : t -> unit
```

```
val write_unlock : t -> unit
```

# Simple Read-write Lock

```
type t

val create : unit -> t

val read_lock : t -> unit
val read_unlock : t -> unit

val write_lock : t -> unit
val write_unlock : t -> unit
```

```
type t = {
  mutable readers : int; (* Current number of active readers *)
  mutable writer : bool; (* Is a writer active? *)
  lock : Mutex.t;
  condition : Condition.t;
}

let create () = {
  readers = 0;
  writer = false;
  lock = Mutex.create ();
  condition = Condition.create ();
}
```

# Simple Read-write Lock

# Simple Read-write Lock

```
let read_lock rwlock =  
  Mutex.lock rwlock.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock rwlock.lock) @@ fun () ->  
    (* Wait while a writer is active *)  
    while rwlock.writer do  
      Condition.wait rwlock.condition rwlock.lock  
    done;  
    (* Increment reader count *)  
    rwlock.readers <- rwlock.readers + 1  
  
let read_unlock rwlock =  
  Mutex.lock rwlock.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock rwlock.lock) @@ fun () ->  
    (* Decrement reader count *)  
    rwlock.readers <- rwlock.readers - 1;  
    (* If no more readers, wake up all waiting threads *)  
    if rwlock.readers = 0 then  
      Condition.broadcast rwlock.condition
```

# Simple Read-write Lock

# Simple Read-write Lock

```
let write_lock rwlock =  
  Mutex.lock rwlock.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock rwlock.lock) @@ fun () ->  
    (* Wait while readers are active OR another writer is active *)  
    while rwlock.readers > 0 || rwlock.writer do  
      Condition.wait rwlock.condition rwlock.lock  
    done;  
    (* Mark writer as active *)  
    rwlock.writer <- true  
  
let write_unlock rwlock =  
  Mutex.lock rwlock.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock rwlock.lock) @@ fun () ->  
    (* Clear writer flag *)  
    rwlock.writer <- false;  
    (* Wake up all waiting threads (readers and writers) *)  
    Condition.broadcast rwlock.condition
```



# Simple Read-write Lock

- Is unfair to writers
  - Can lead to starvation of writers if readers keep coming in

# Fair Read-write Locks

```
type t = {  
  mutable read_acquires : int; (* Total read locks acquired *)  
  mutable read_releases : int; (* Total read locks released *)  
  mutable writer : bool;      (* Is a writer active? *)  
  lock : Mutex.t;  
  condition : Condition.t;  
}  
  
let create () = {  
  read_acquires = 0;  
  read_releases = 0;  
  writer = false;  
  lock = Mutex.create ();  
  condition = Condition.create ();  
}
```

# Fair Read-write Locks

```
let read_lock rwlock =  
  Mutex.lock rwlock.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock rwlock.lock) @@ fun () ->  
    (* Wait while a writer is active BEFORE incrementing counter *)  
    while rwlock.writer do  
      Condition.wait rwlock.condition rwlock.lock  
    done;  
    (* Only now increment acquisition counter *)  
    rwlock.read_acquires <- rwlock.read_acquires + 1  
  
let read_unlock rwlock =  
  Mutex.lock rwlock.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock rwlock.lock) @@ fun () ->  
    (* Increment release counter *)  
    rwlock.read_releases <- rwlock.read_releases + 1;  
    (* If all acquired reads have been released, wake up waiting writers *)  
    if rwlock.read_acquires = rwlock.read_releases then  
      Condition.broadcast rwlock.condition
```

# Fair Read-write Locks

```
let write_lock rwlock =  
  Mutex.lock rwlock.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock rwlock.lock) @@ fun () ->  
    (* Phase 1: Wait for no active writer *)  
    while rwlock.writer do  
      Condition.wait rwlock.condition rwlock.lock  
    done;  
    (* Claim writer status to block new readers *)  
    rwlock.writer <- true;  
    (* Phase 2: Wait for existing readers to drain *)  
    while rwlock.read_acquires <> rwlock.read_releases do  
      Condition.wait rwlock.condition rwlock.lock  
    done
```

```
let write_unlock rwlock =  
  Mutex.lock rwlock.lock;  
  Fun.protect ~finally:(fun () -> Mutex.unlock rwlock.lock) @@ fun () ->  
    (* Clear writer flag *)  
    rwlock.writer <- false;  
    (* Wake up all waiting threads *)  
    Condition.broadcast rwlock.condition
```

# When should we use monitors?

- Monitors are complementary to spin-locks
- Spin-locks are good when the *expected wait time is small*
- Monitors are good when the *expected wait time is large*
  - Expensive to context-switch
- OS mutexes already spin for a little while before going to block

# Other Synchronisation mechanisms

- **Monitors** are generally the most popular synchronisation mechanism used widely
- A **semaphore** allows at most  $n \geq 1$  threads to concurrently be in the critical section
  - Edsger Dijkstra (the same as in Dijkstra's algorithm) in 1963
  - A **Mutex** is a semaphore with  $n = 1$

**Fin**