Whatever can go wrong
will go wrong.

attributed to Edward A. Murphy

Murphy was an optimist.

authors of lock-free programs

3. LOCK FREE KERNEL

Literature

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

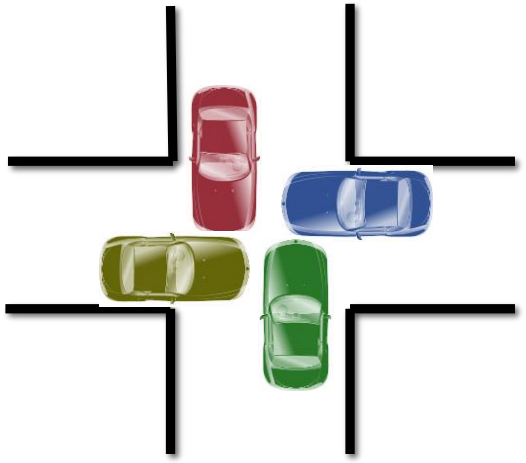
Florian Negele. *Combining Lock-Free Programming with Cooperative Multitasking for a Portable Multiprocessor Runtime System.* ETH-Zürich, 2014. http://dx.doi.org/10.3929/ethz-a-010335528

A substantial part of the following material is based on Florian Negele's Thesis.

Florian Negele, Felix Friedrich, Suwon Oh and Bernhard Egger, *On the Design and Implementation of an Efficient Lock-Free Scheduler,* 19th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) 2015.

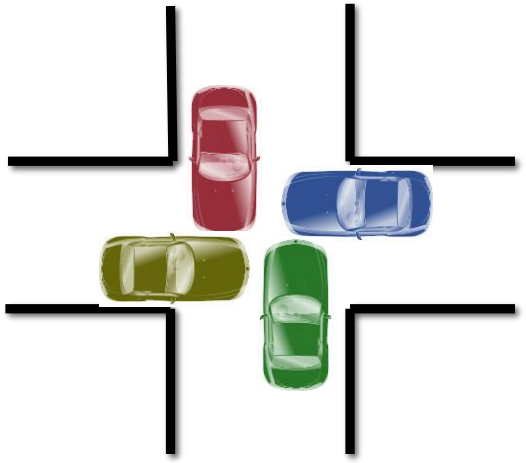
Problems with Locks

Problems with Locks

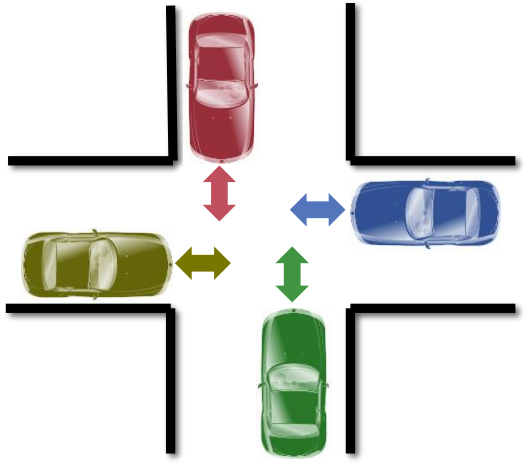


Deadlock

Problems with Locks

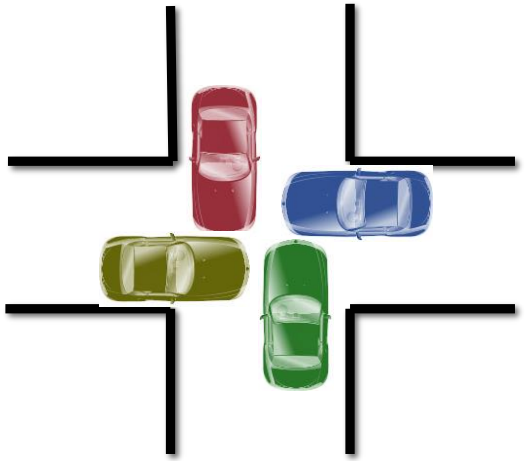


Deadlock

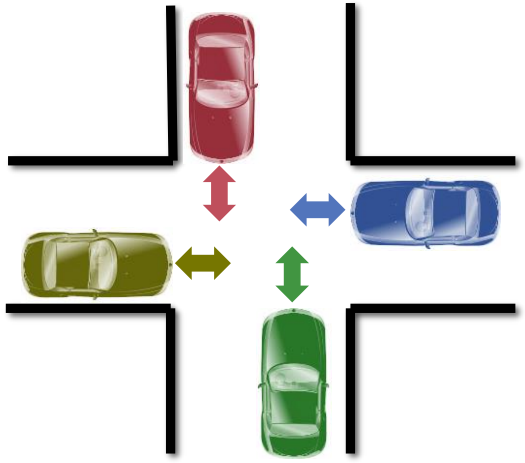


Livelock

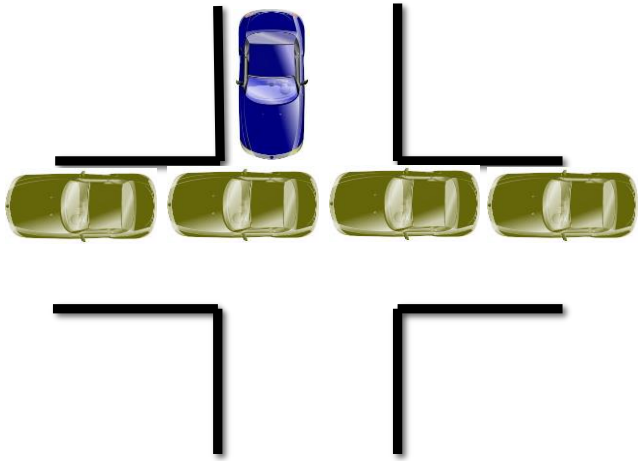
Problems with Locks



Deadlock

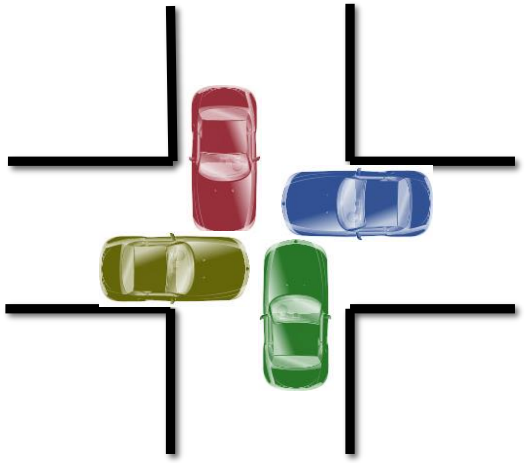


Livelock

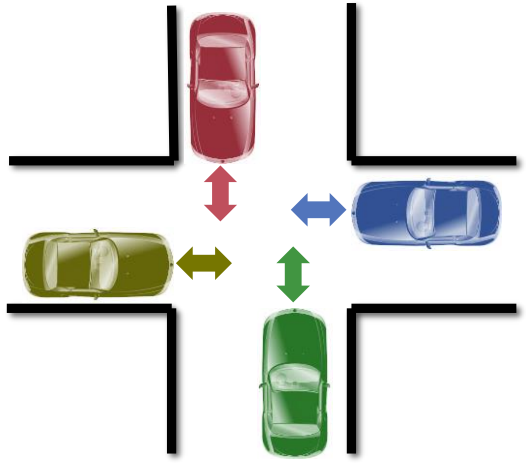


Starvation

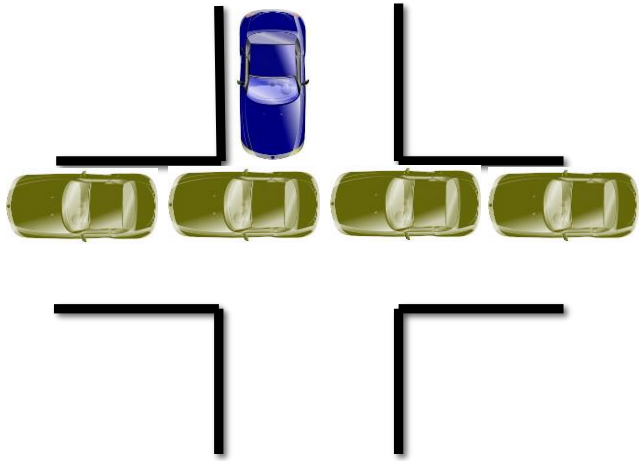
Problems with Locks



Deadlock



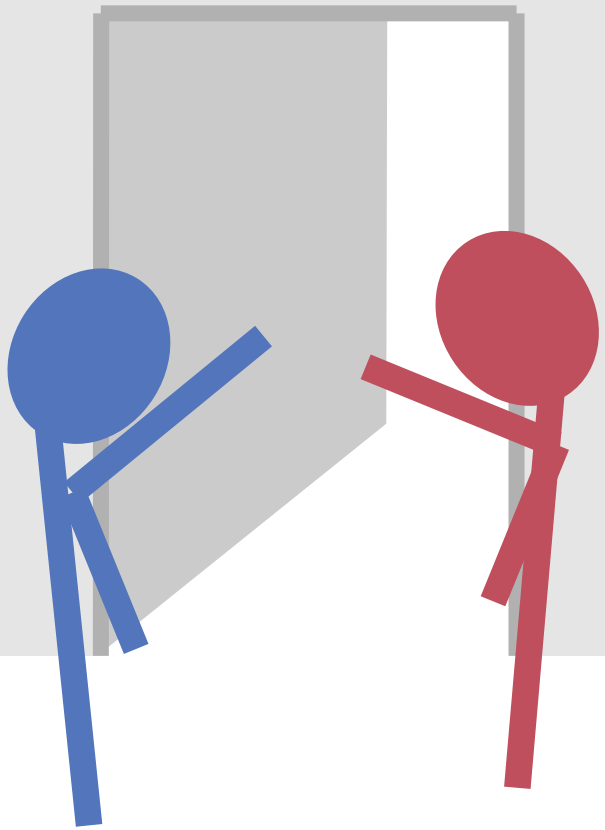
Livelock



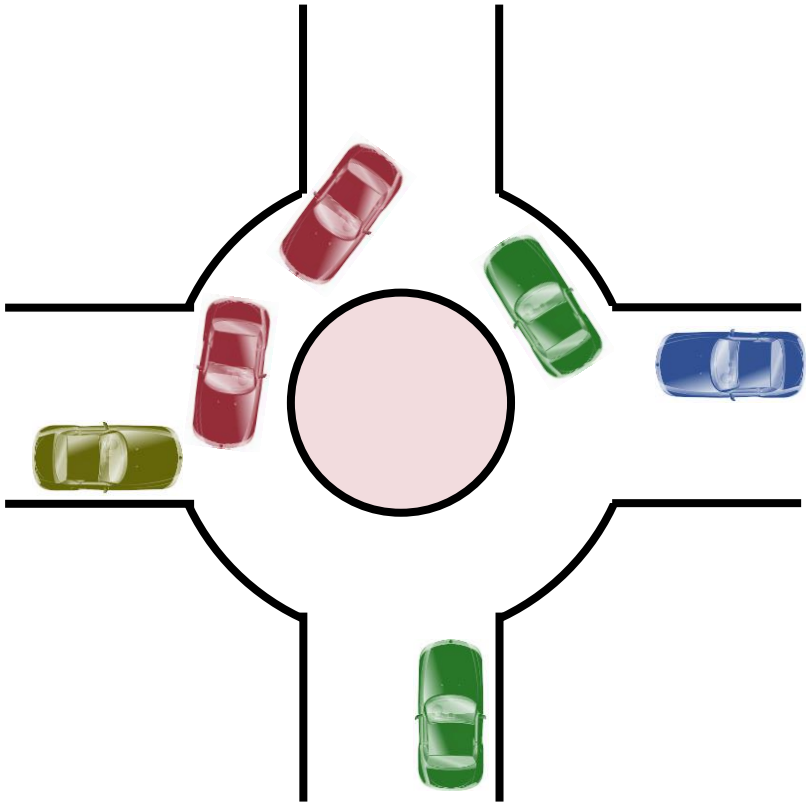
Starvation

Parallelism? Progress Guarantees? Reentrancy? Granularity? Fault Tolerance?

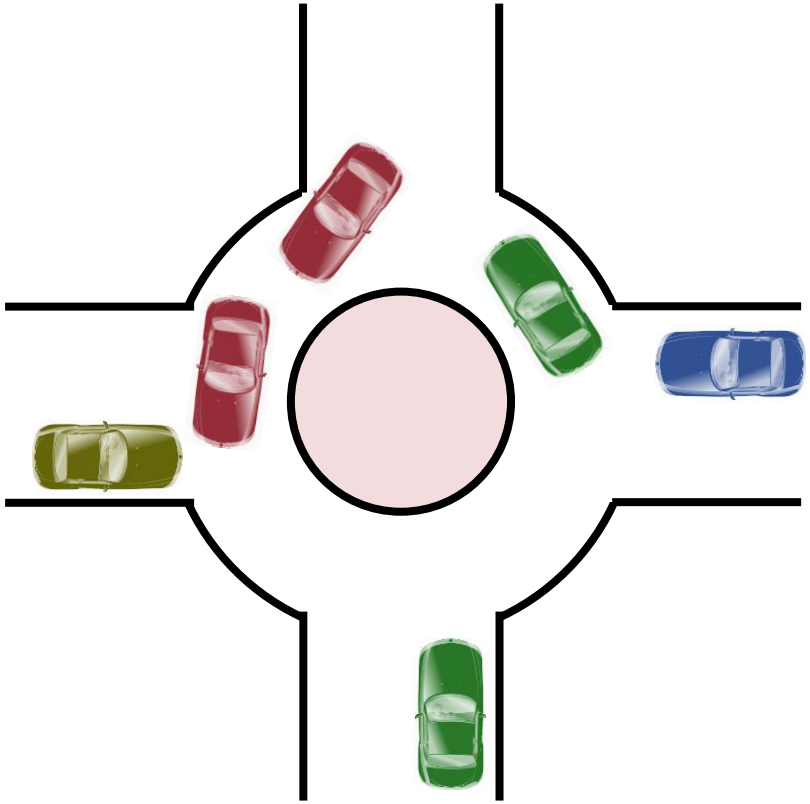
Politelock

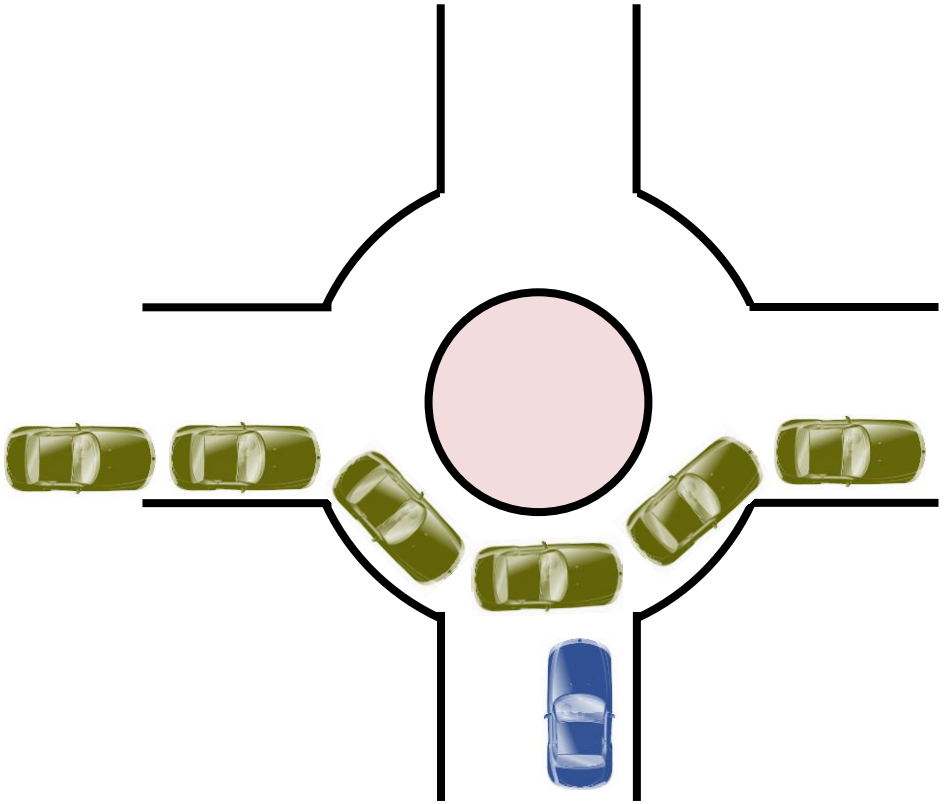


Lock-Free



Lock-Free





Definitions

Lock-freedom: at least one algorithm makes progress even if other algorithms run concurrently, fail or get suspended.
Implies system-wide progress but not freedom from starvation.

Definitions

Lock-freedom: at least one algorithm makes progress even if other algorithms run concurrently, fail or get suspended.
Implies system-wide progress but not freedom from starvation.

Wait-freedom: each algorithm eventually makes progress.
Implies freedom from starvation.

Definitions

Lock-freedom: at least one algorithm makes progress even if other algorithms run concurrently, fail or get suspended.
Implies system-wide progress but not freedom from starvation.

Wait-freedom: each algorithm eventually makes progress.
Implies freedom from starvation.

Progress Conditions

	Blocking	Non-Blocking
Someone make progress	Deadlock-free	**Lock-free**
Everyone makes progress	Starvation-free	Wait-free

Goals

Goals

Lock Freedom

- Progress Guarantees
- Reentrant Algorithms

Goals

Portability

- Hardware Independence
- Simplicity, Maintenance

Guiding principles

1. Keep things **simple**
2. Exclusively employ **non-blocking** algorithms in the system

Guiding principles

1. Keep things **simple**
2. Exclusively employ **non-blocking** algorithms in the system

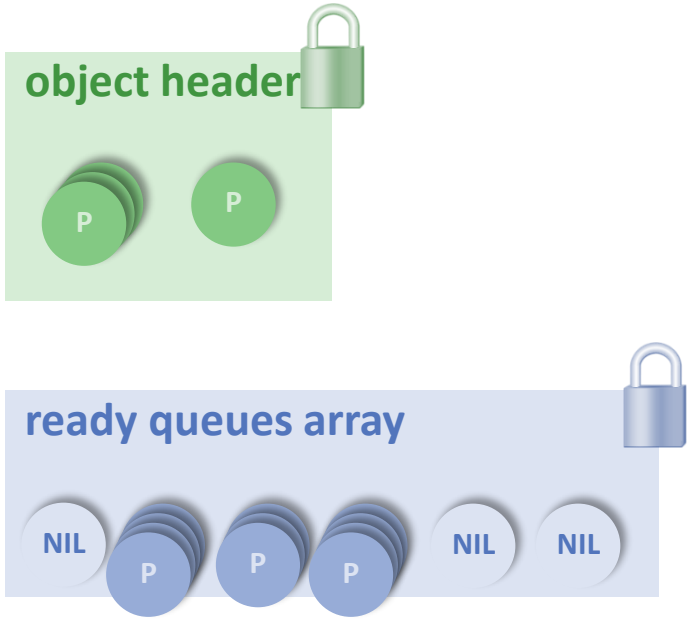
→ Use **implicit cooperative multitasking**

→ no virtual memory

→ limits in optimization

Where are the Locks in the Kernel?

Scheduling Queues / Heaps



Memory Management

CAS (again)

int CAS (memref a, int old, int new)

atomic

CAS (again)

- Compare **old** with data at memory location

int CAS (memref a, int old, int new)

atomic

previous = mem[a];

CAS (again)

- Compare **old** with data at memory location

- If and only if data at memory equals **old** overwrite data with **new**

int CAS (memref a, int old, int new)

atomic

```
previous = mem[a];
if (old == previous)
    Mem[a] = new;
```

CAS (again)

- Compare **old** with data at memory location
- If and only if data at memory equals **old** overwrite data with **new**
- Return previous memory value

int CAS (memref a, int old, int new)

atomic

```
previous = mem[a];
if (old == previous)
        Mem[a] = new;
return previous;
```

CAS (again)

- Compare **old** with data at memory location
- If and only if data at memory equals **old** overwrite data with **new**
- Return previous memory value

```
int CAS (memref a, int old, int new)
atomic {
    previous = mem[a];
    if (old == previous)
        Mem[a] = new;
    return previous;
}
```

CAS is implemented wait-free(!) by hardware.

Memory Model for Lockfree Active Oberon

Memory Model for Lockfree Active Oberon

Only **two rules**

Memory Model for Lockfree Active Oberon

Only **two rules**

1. **Data shared** between two or more activities at the same time has to be **protected using exclusive blocks** unless the data is read or modified using the compare-and-swap operation

Memory Model for Lockfree Active Oberon

Only **two rules**

1. **Data shared** between two or more activities at the same time has to be **protected using exclusive blocks** unless the data is read or modified using the compare-and-swap operation
2. Changes to shared data **visible to other activities after leaving an exclusive block or executing a compare-and-swap** operation.

   Implementations are free to reorder all other memory accesses as long as their effect equals a sequential execution within a single activity.

Inbuilt CAS

- CAS instruction as statement of the language

```
PROCEDURE CAS(VAR variable, old, new: BaseType): BaseType
```

  - Operation executed atomically, result visible instantaneously to other processes
  - CAS(variable, x, x) constitutes an atomic read

Inbuilt CAS

- CAS instruction as statement of the language

```
PROCEDURE CAS(VAR variable, old, new: BaseType): BaseType
```

  - Operation executed atomically, result visible instantaneously to other processes
  - CAS(variable, x, x) constitutes an atomic read
- Compiler required to implement CAS as a synchronisation barrier
  - Portability, even for non-blocking algorithms
  - Consistent view on shared data, even for systems that represent words using bytes

Simple Example: Non-blocking counter

```
PROCEDURE Increment(VAR counter: LONGINT): LONGINT;
VAR previous, value: LONGINT;
BEGIN
   REPEAT
      previous := CAS(counter,0,0);
      value := CAS(counter, previous, previous + 1);
   UNTIL value = previous;
   return previous;
END Increment;
```

Lock-Free Programming

Lock-Free Programming

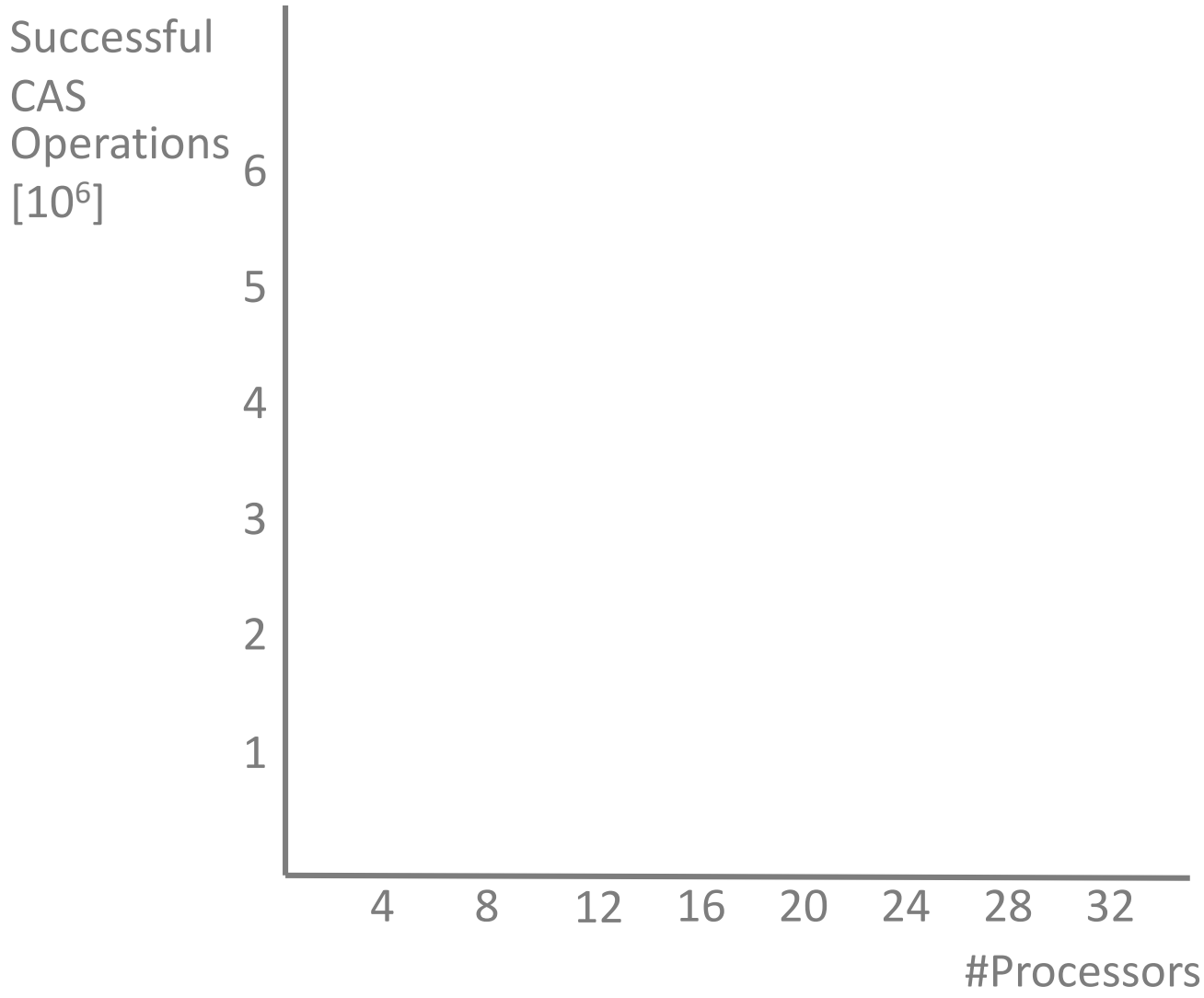
Performance of CAS

- on the H/W level, CAS triggers a memory barrier
- performance suffers with increasing number of contenders to the same variable

Lock-Free Programming

Performance of CAS

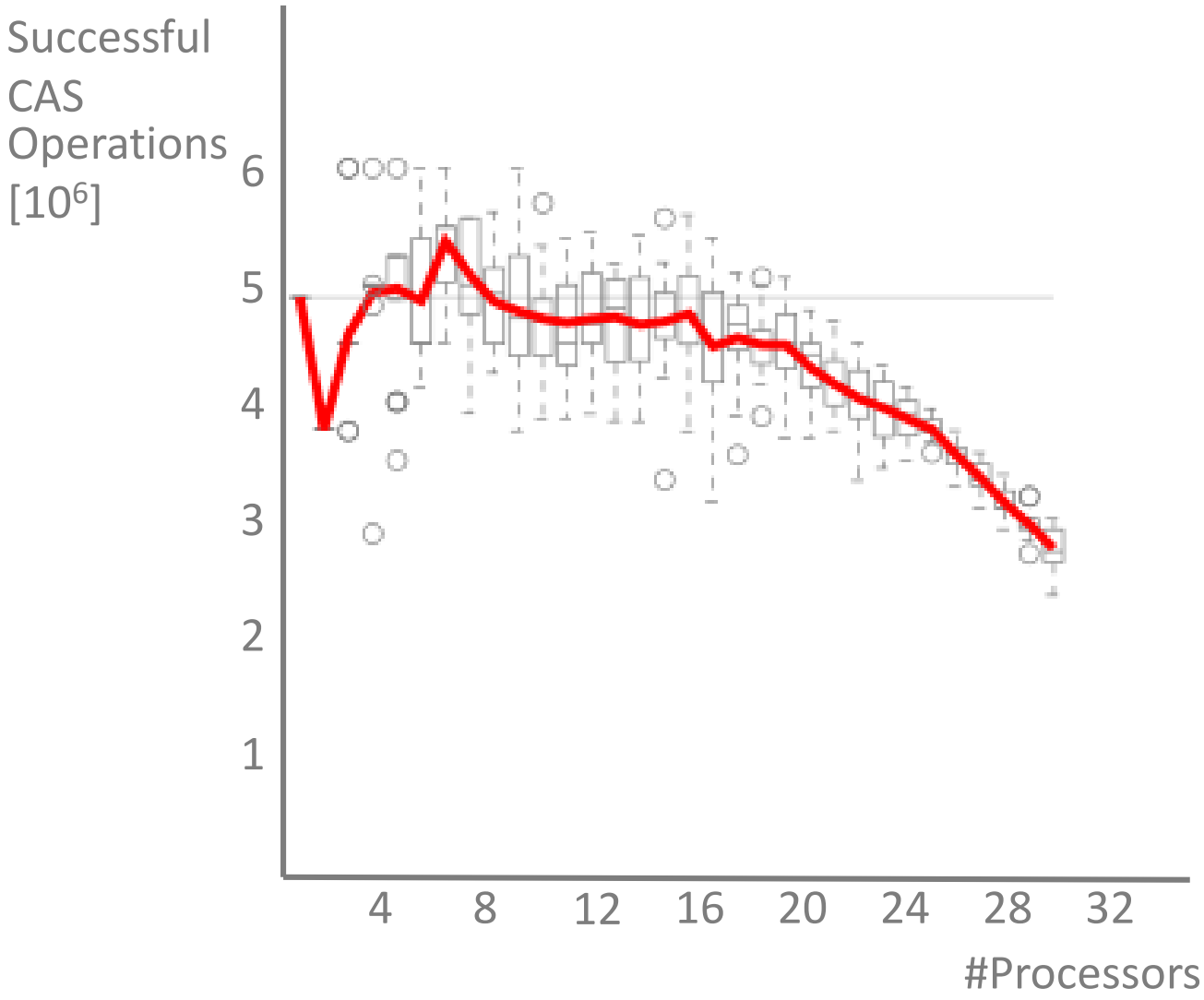
- on the H/W level, CAS triggers a memory barrier
- performance suffers with increasing number of contenders to the same variable



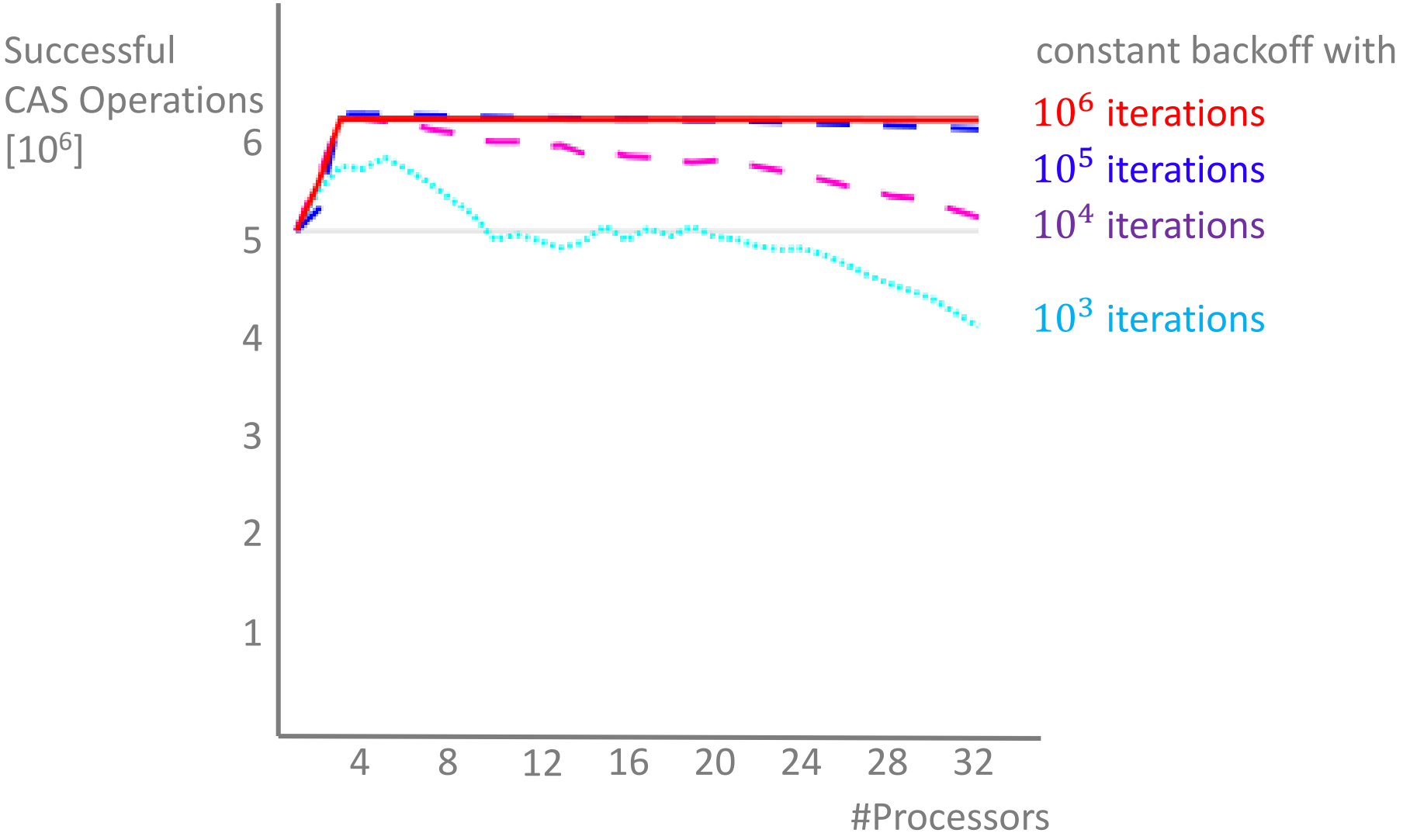
Lock-Free Programming

Performance of CAS

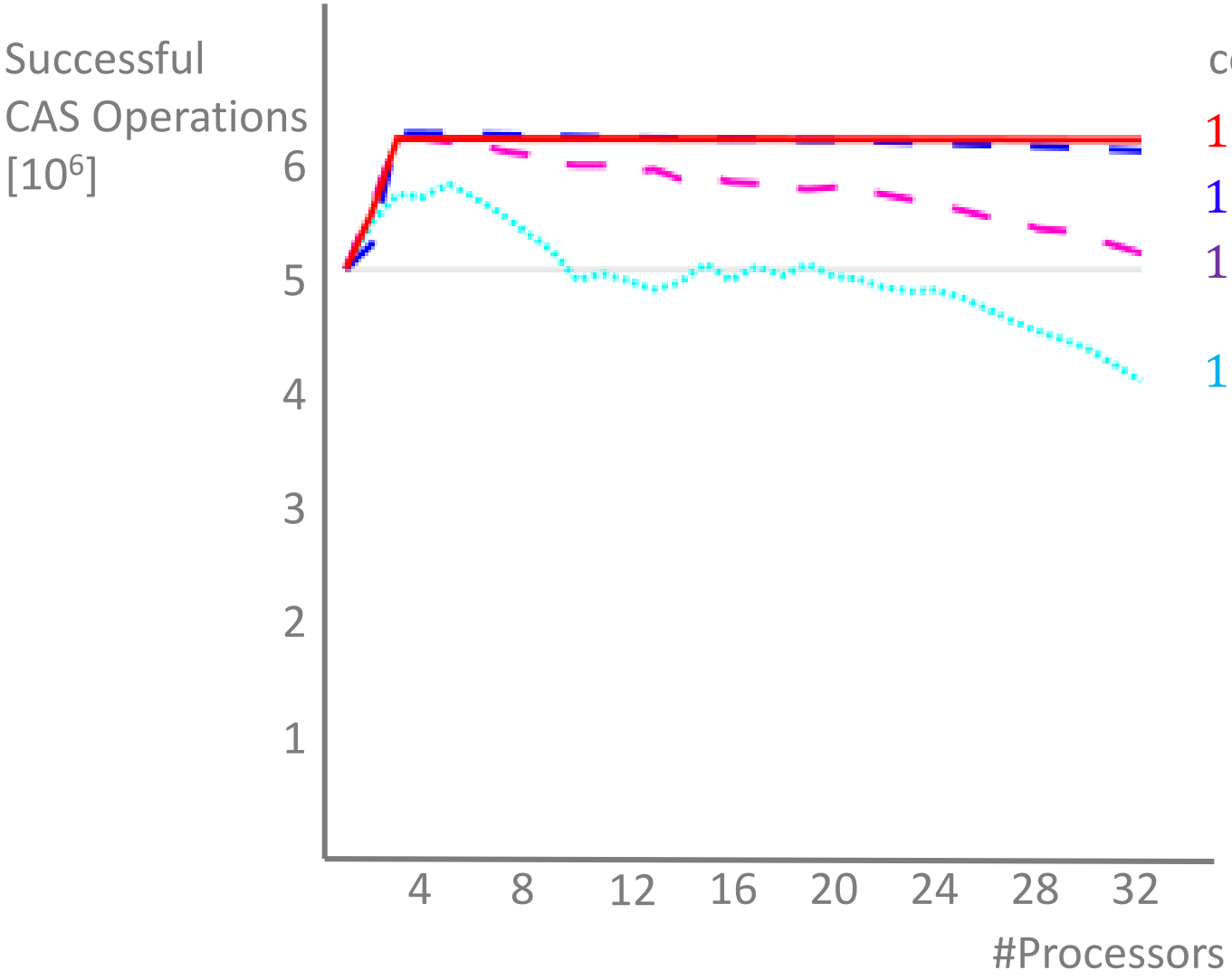
- on the H/W level, CAS triggers a memory barrier
- performance suffers with increasing number of contenders to the same variable



CAS with backoff



CAS with backoff



constant backoff with

10^6 iterations

10^5 iterations

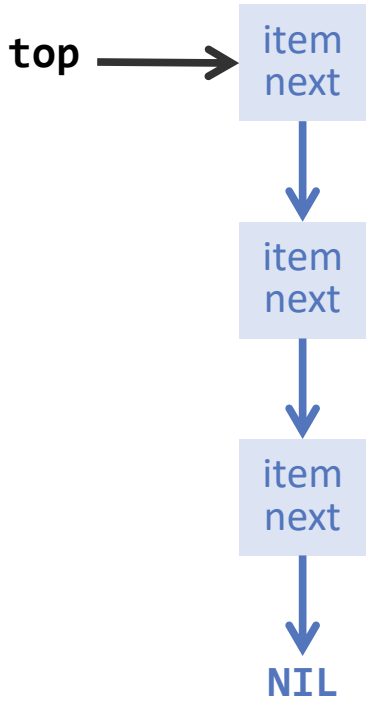
10^4 iterations

10^3 iterations

```
LOOP
    value := CAS (counter, 0, 0);
    IF CAS (counter, v, v+1) = v THEN
        EXIT
    END;
    CPU.Backoff;
END;
```

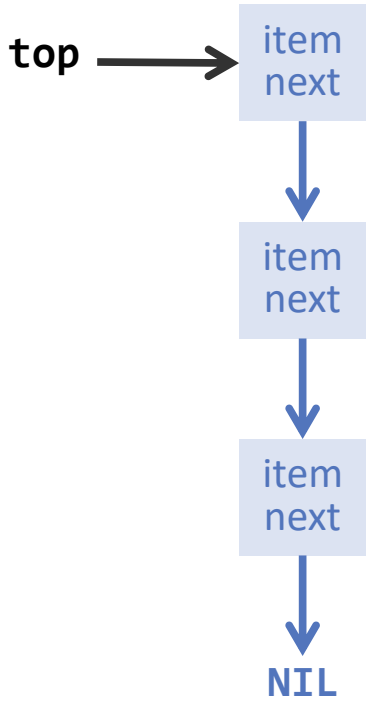
Stack

```
Node = POINTER TO RECORD
  item: Object;
  next: Node;
END;

Stack = OBJECT
VAR top: Node;
  PROCEDURE Pop(VAR head: Node): BOOLEAN;
  PROCEDURE Push(head: Node);
END;
```



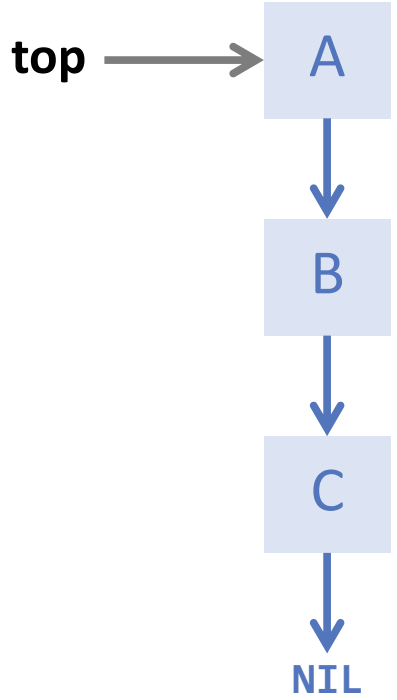
Stack -- Blocking

```
PROCEDURE Push(node: Node): BOOLEAN;
BEGIN{EXCLUSIVE}
   node.next := top;
   top := node;
END Push;

PROCEDURE Pop(VAR head: Node): BOOLEAN;
VAR next: Node;
BEGIN{EXCLUSIVE}
   head := top;
   IF head = NIL THEN
      RETURN FALSE
   ELSE
      top := head.next;
      RETURN TRUE;
   END;
END Pop;
```



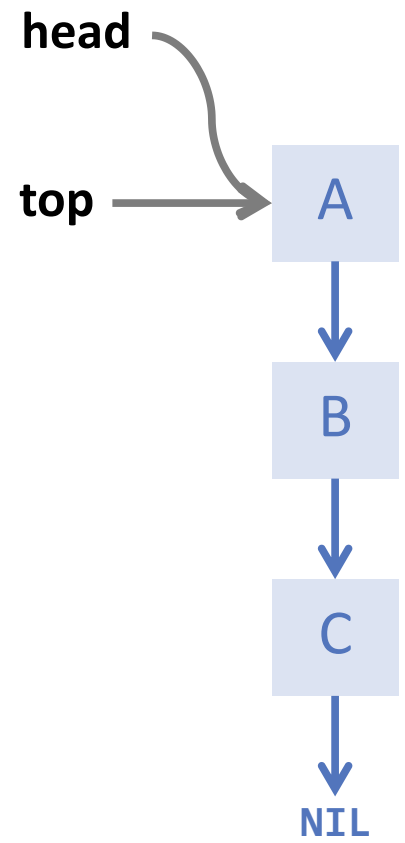
Stack -- Lockfree

```
PROCEDURE Pop(VAR head: Node): BOOLEAN;
VAR next: Node;
BEGIN




END Pop;
```



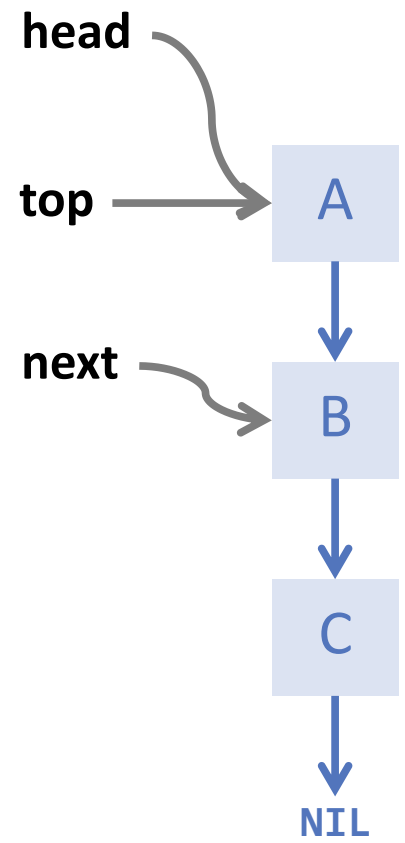
Stack -- Lockfree

```
PROCEDURE Pop(VAR head: Node): BOOLEAN;
VAR next: Node;
BEGIN

     head := CAS(top, NIL, NIL);
     IF head = NIL THEN
        RETURN FALSE
     END;
     next := head.next;




END Pop;
```



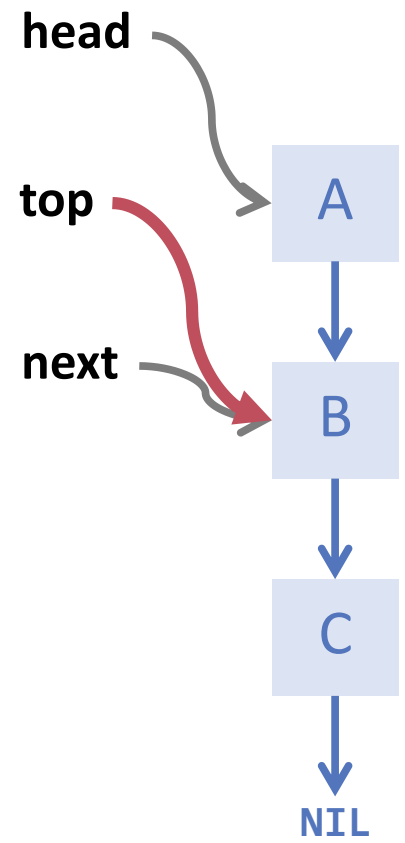
Stack -- Lockfree

```
PROCEDURE Pop(VAR head: Node): BOOLEAN;
VAR next: Node;
BEGIN

     head := CAS(top, NIL, NIL);
     IF head = NIL THEN
        RETURN FALSE
     END;
     next := head.next;




END Pop;
```



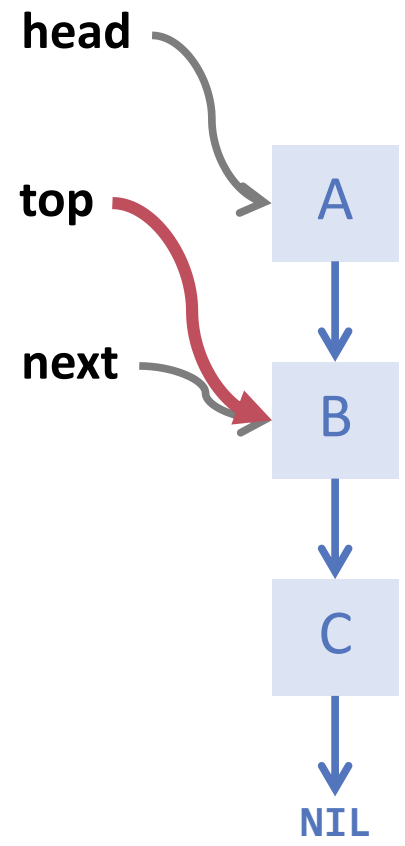
Stack -- Lockfree

```
PROCEDURE Pop(VAR head: Node): BOOLEAN;
VAR next: Node;
BEGIN

      head := CAS(top, NIL, NIL);
      IF head = NIL THEN
         RETURN FALSE
      END;
      next := head.next;
      IF CAS(top, head, next) = head THEN
         RETURN TRUE
      END;


END Pop;
```



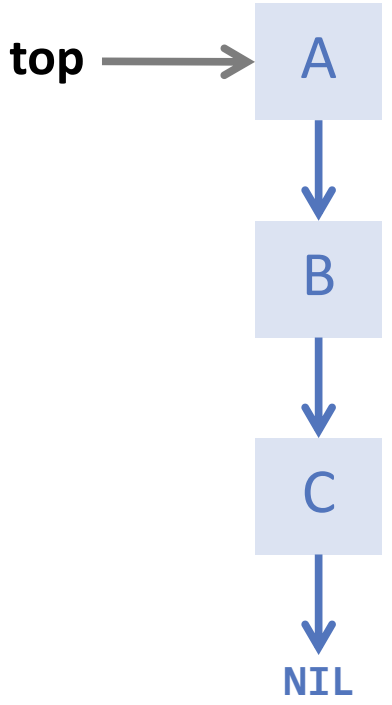
Stack -- Lockfree

```
PROCEDURE Pop(VAR head: Node): BOOLEAN;
VAR next: Node;
BEGIN
   LOOP
      head := CAS(top, NIL, NIL);
      IF head = NIL THEN
         RETURN FALSE
      END;
      next := head.next;
      IF CAS(top, head, next) = head THEN
         RETURN TRUE
      END;
      CPU.Backoff
   END;
END Pop;
```



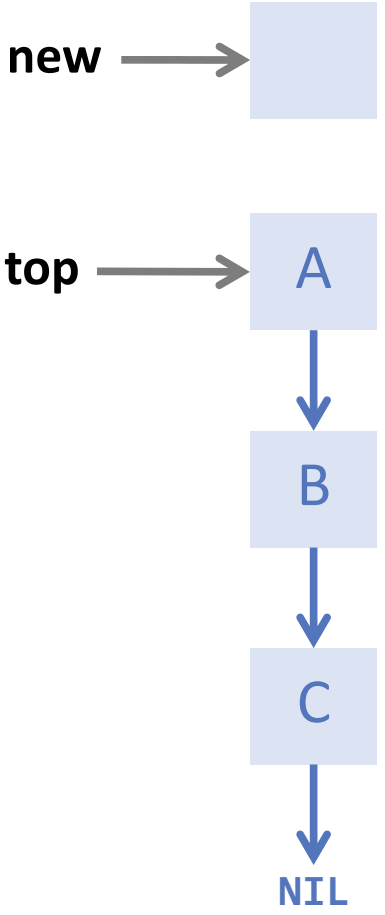
Stack -- Lockfree

```
PROCEDURE Push(new: Node);
BEGIN




END Push;
```



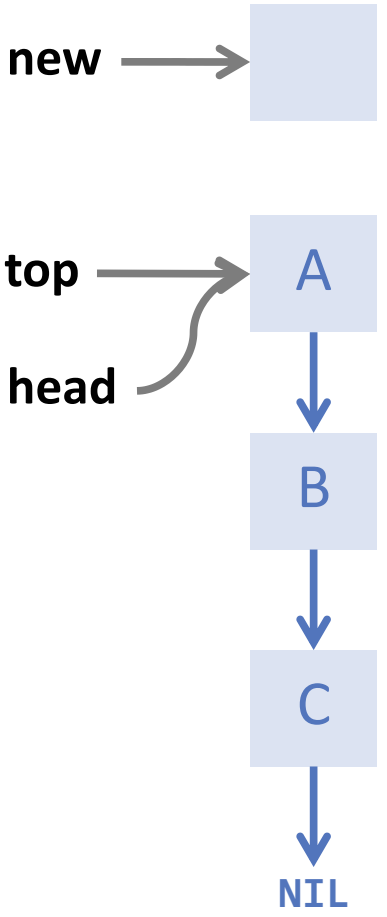
Stack -- Lockfree

```
PROCEDURE Push(new: Node);
BEGIN




END Push;
```



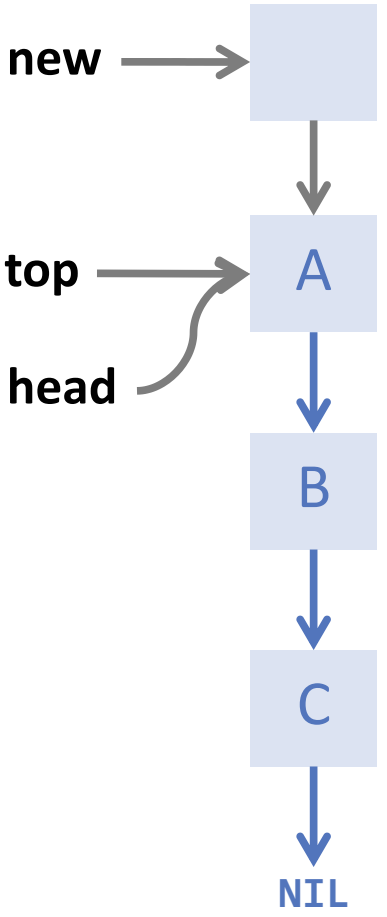
Stack -- Lockfree

```
PROCEDURE Push(new: Node);
BEGIN

      head := CAS(top, NIL, NIL);
      new.next := head;




END Push;
```



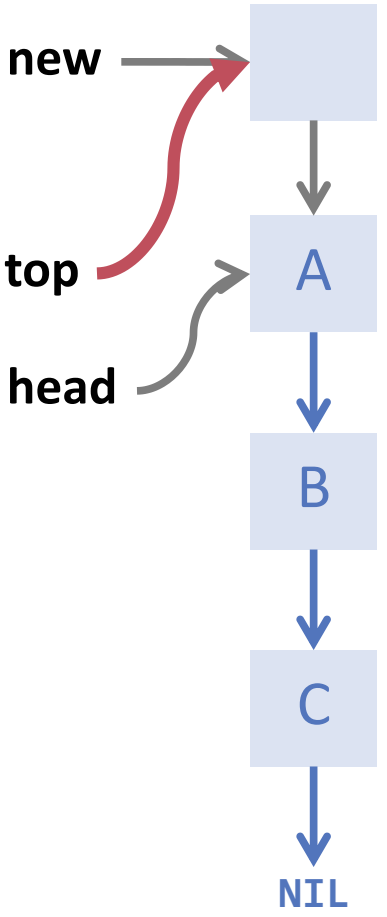
Stack -- Lockfree

```
PROCEDURE Push(new: Node);
BEGIN

      head := CAS(top, NIL, NIL);
      new.next := head;




END Push;
```



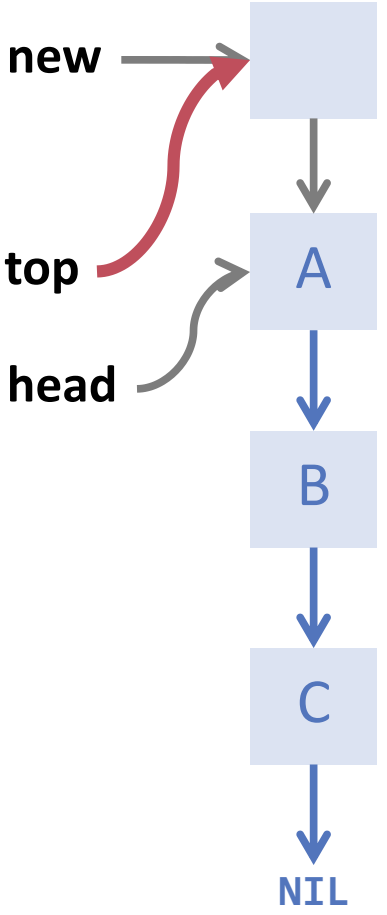
Stack -- Lockfree

```
PROCEDURE Push(new: Node);
BEGIN

      head := CAS(top, NIL, NIL);
      new.next := head;
      IF CAS(top, head, new) = head THEN
         EXIT
      END;


END Push;
```



Stack -- Lockfree

```
PROCEDURE Push(new: Node);
BEGIN
   LOOP
      head := CAS(top, NIL, NIL);
      new.next := head;
      IF CAS(top, head, new) = head THEN
         EXIT
      END;
      CPU.Backoff;
   END;
END Push;
```



Node Reuse

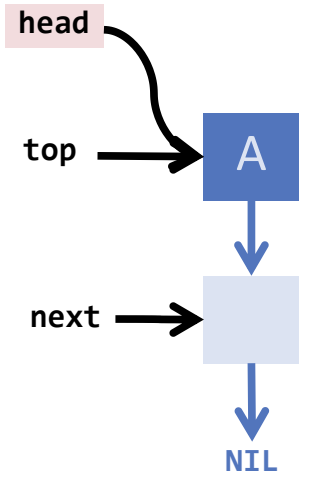
Assume we do not want to allocate a new node for each Push and maintain a Node-pool instead. Does this work?

NO ! WHY NOT?

ABA Problem

ABA Problem

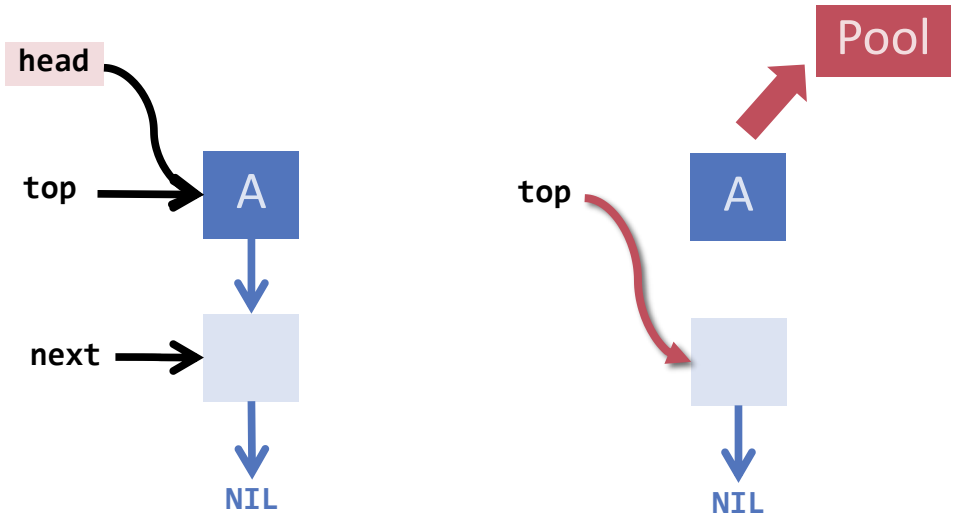
Thread X
in the middle
of pop: after read
but before CAS



ABA Problem

Thread X
in the middle
of pop: after read
but before CAS

Thread Y
pops A



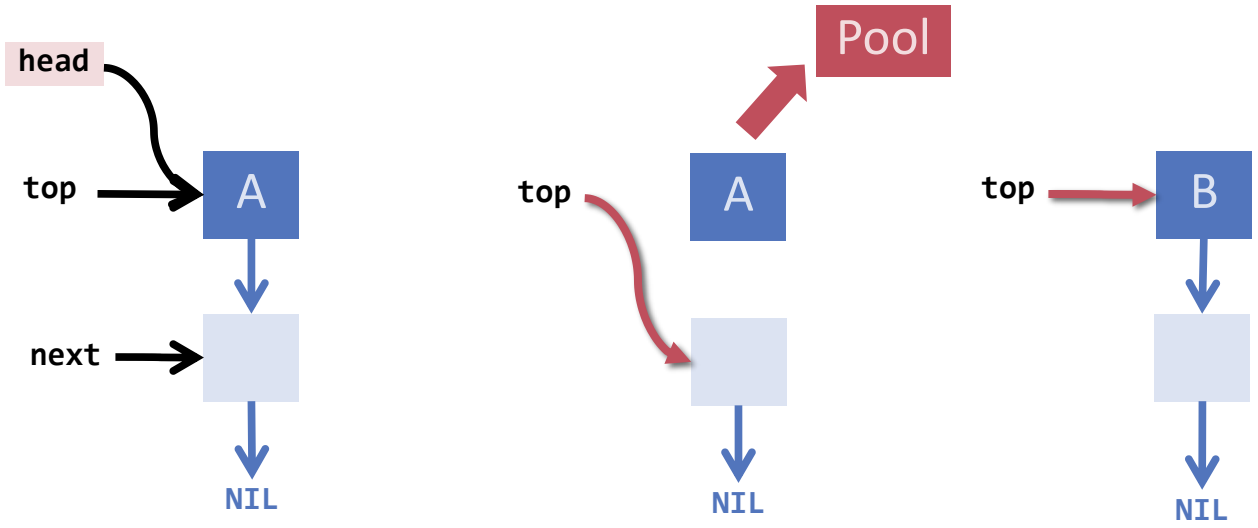
time

ABA Problem

Thread X
in the middle
of pop: after read
but before CAS

Thread Y
pops A

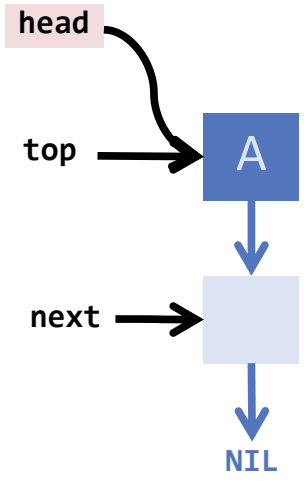
Thread Z
pushes B



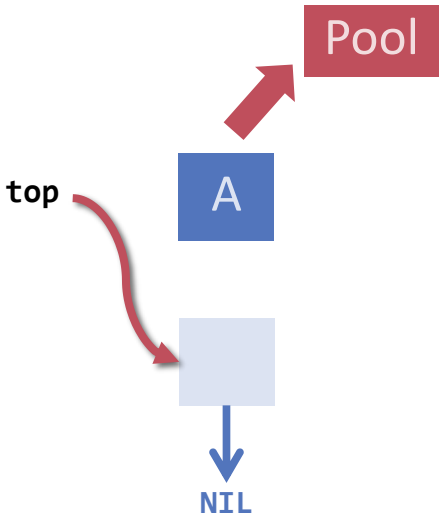
time

ABA Problem

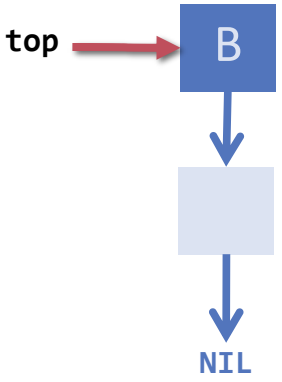
Thread X
in the middle
of pop: after read
but before CAS



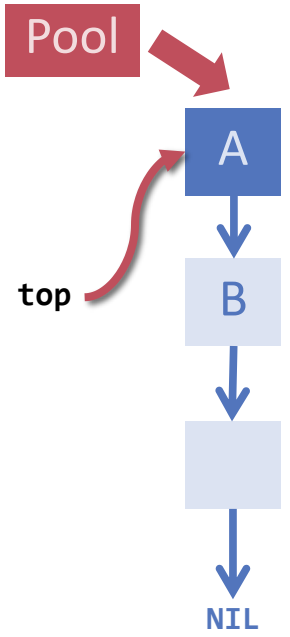
Thread Y
pops A



Thread Z
pushes B



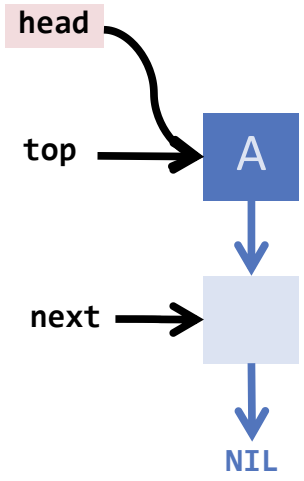
Thread Z'
pushes A



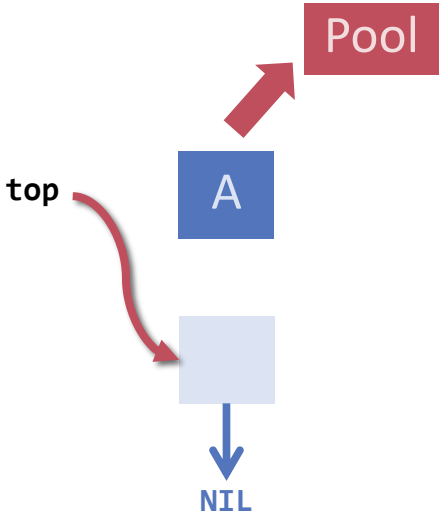
time

ABA Problem

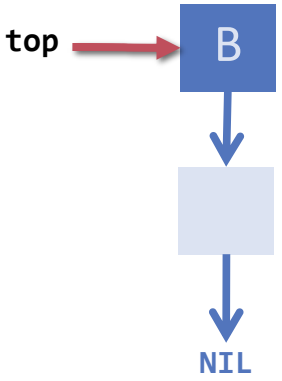
Thread X
in the middle
of pop: after read
but before CAS



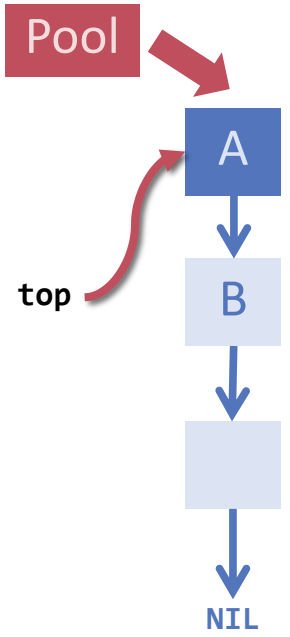
Thread Y
pops A



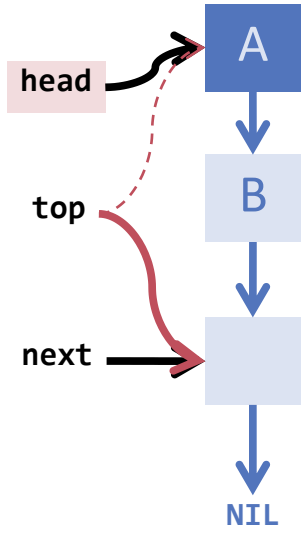
Thread Z
pushes B



Thread Z'
pushes A



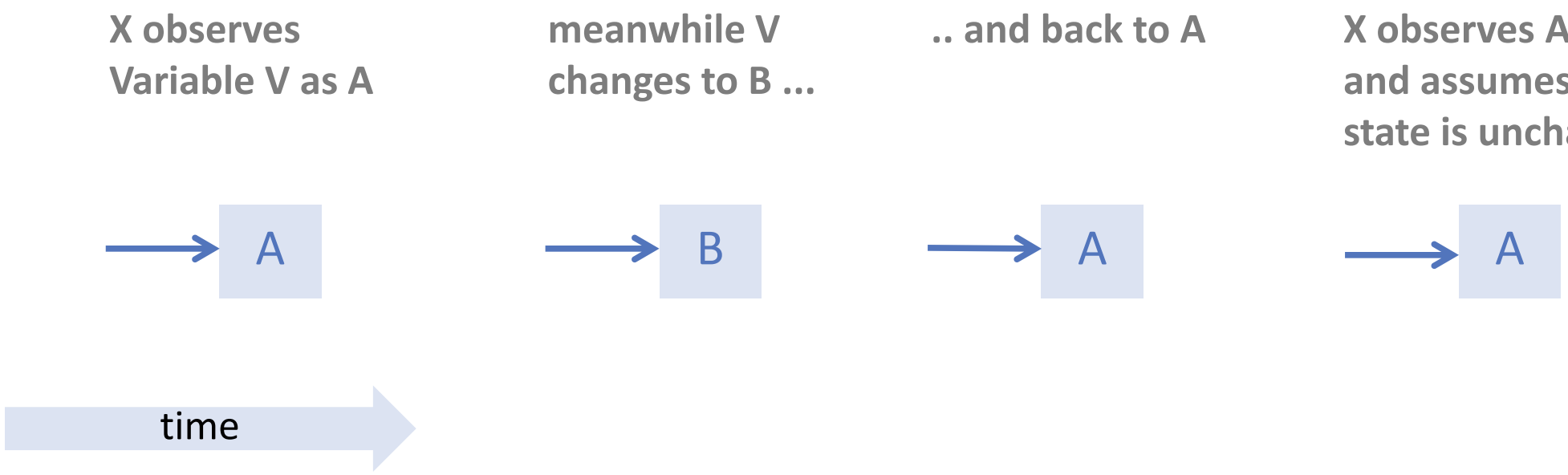
Thread X
completes pop



time

The **ABA**-Problem

"The ABA problem ... occurs when one activity fails to recognise that a single memory location was modified temporarily by another activity and therefore erroneously assumes that the overal state has not been changed."



How to solve the ABA problem?

How to solve the ABA problem?

- DCAS (double compare and swap)
  - not available on most platforms

How to solve the ABA problem?

- DCAS (double compare and swap)
  - not available on most platforms
- Hardware transactional memory
  - not available on most platforms
  - memory restrictions

How to solve the ABA problem?

- DCAS (double compare and swap)
  - not available on most platforms
- Hardware transactional memory
  - not available on most platforms
  - memory restrictions
- Garbage Collection
  - relies on the existence of a GC
  - impossible to use in the inner of a runtime kernel
  - can you implement a lock-free garbage collector relying on garbage collection?

How to solve the ABA problem?

- DCAS (double compare and swap)
  - not available on most platforms
- Hardware transactional memory
  - not available on most platforms
  - memory restrictions
- Garbage Collection
  - relies on the existence of a GC
  - impossible to use in the inner of a runtime kernel
  - can you implement a lock-free garbage collector relying on garbage collection?
- **Pointer Tagging**
  - does not cure the problem, rather delay it
  - can be practical

How to solve the ABA problem?

- DCAS (double compare and swap)
  - not available on most platforms
- Hardware transactional memory
  - not available on most platforms
  - memory restrictions
- Garbage Collection
  - relies on the existence of a GC
  - impossible to use in the inner of a runtime kernel
  - can you implement a lock-free garbage collector relying on garbage collection?
- **Pointer Tagging**
  - does not cure the problem, rather delay it
  - can be practical
- **Hazard Pointers**

Pointer Tagging

ABA problem usually occurs with CAS on *pointers*

Aligned addresses (values of pointers) make some bits available for *pointer tagging.*

Pointer Tagging

ABA problem usually occurs with CAS on *pointers*

Aligned addresses (values of pointers) make some bits available for *pointer tagging.*

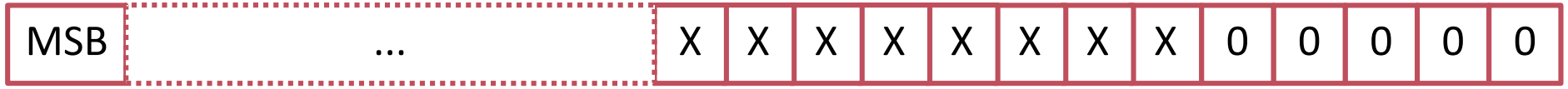
Example: pointer aligned modulo 32 → 5 bits available for tagging

Pointer Tagging

ABA problem usually occurs with CAS on *pointers*

Aligned addresses (values of pointers) make some bits available for *pointer tagging.*

Example: pointer aligned modulo 32 → 5 bits available for tagging

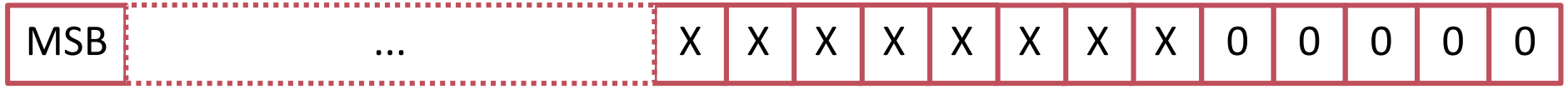


Pointer Tagging

ABA problem usually occurs with CAS on *pointers*

Aligned addresses (values of pointers) make some bits available for *pointer tagging.*

Example: pointer aligned modulo 32 → 5 bits available for tagging



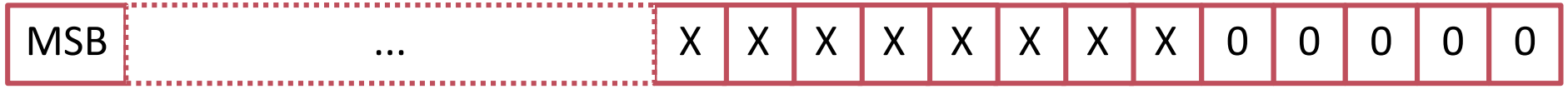
Each time a pointer is stored in a data structure, the tag is increased by one. Access to a data structure via address x – x mod 32

Pointer Tagging

ABA problem usually occurs with CAS on *pointers*

Aligned addresses (values of pointers) make some bits available for *pointer tagging.*

Example: pointer aligned modulo 32 → 5 bits available for tagging



Each time a pointer is stored in a data structure, the tag is increased by one. Access to a data structure via address x – x mod 32

This makes the ABA problem very much less probable because now 32 versions of each pointer exist.

Hazard Pointers

Hazard Pointers

The ABA problem stems from reuse of a pointer P that has been read by some thread X but not yet written with CAS by the same thread. Modification takes place meanwhile by some other thread Y.

Hazard Pointers

The ABA problem stems from reuse of a pointer P that has been read by some thread X but not yet written with CAS by the same thread. Modification takes place meanwhile by some other thread Y.

Idea to solve:

Hazard Pointers

The ABA problem stems from reuse of a pointer P that has been read by some thread X but not yet written with CAS by the same thread. Modification takes place meanwhile by some other thread Y.

Idea to solve:

- Before X reads P, it marks it **hazarduous** by entering it in a thread-dedicated slot of the n (n= number threads) slots of an array associated with the data structure (e.g. the stack)

Hazard Pointers

The ABA problem stems from reuse of a pointer P that has been read by some thread X but not yet written with CAS by the same thread. Modification takes place meanwhile by some other thread Y.

Idea to solve:

- Before X reads P, it marks it **hazarduous** by entering it in a thread-dedicated slot of the n (n= number threads) slots of an array associated with the data structure (e.g. the stack)
- When finished (after the CAS), process X removes P from the array

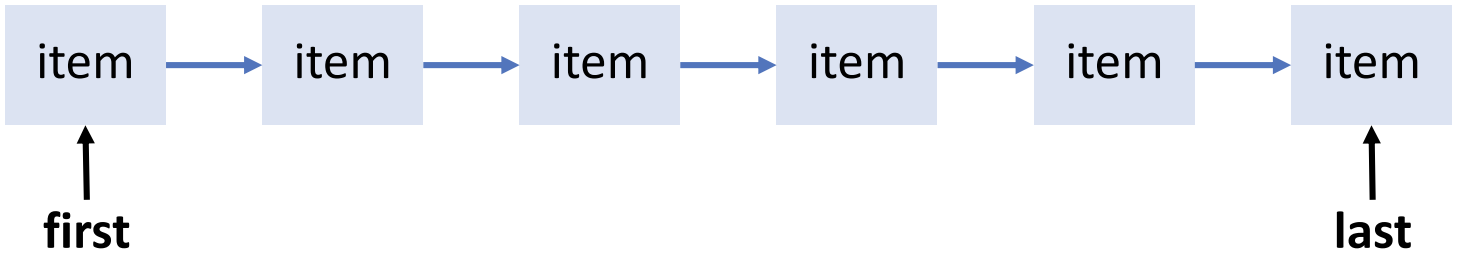
Hazard Pointers

The ABA problem stems from reuse of a pointer P that has been read by some thread X but not yet written with CAS by the same thread. Modification takes place meanwhile by some other thread Y.

Idea to solve:

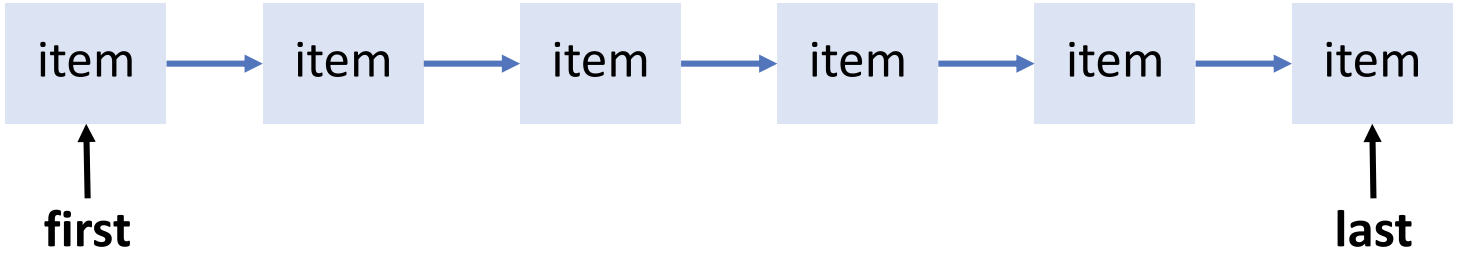
- Before X reads P, it marks it **hazarduous** by entering it in a thread-dedicated slot of the n (n= number threads) slots of an array associated with the data structure (e.g. the stack)
- When finished (after the CAS), process X removes P from the array
- Before a process Y tries to reuse P, it checks all entries of the hazard array

Unbounded Queue (FIFO)



Enqueue

case last != NIL

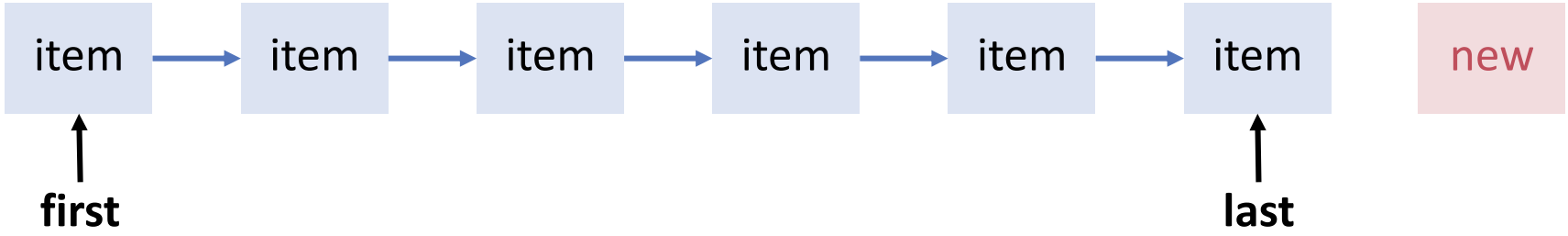


case last = NIL

first **last**

Enqueue

case last != NIL

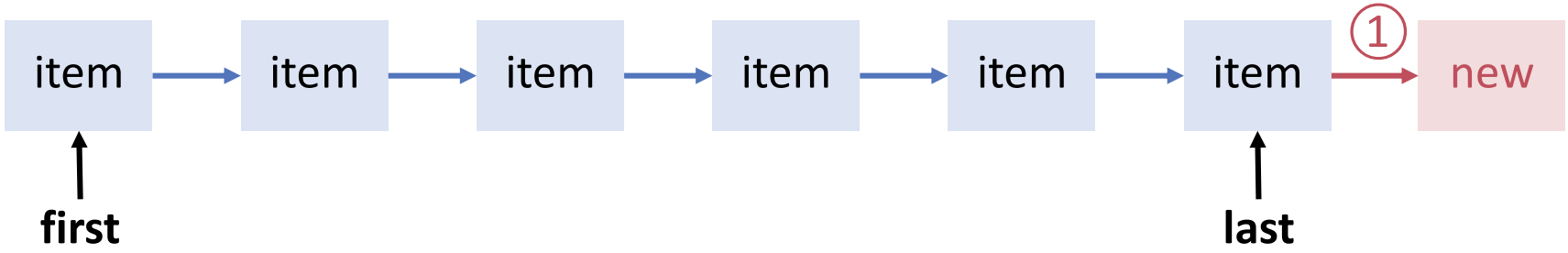


case last = NIL

first **last**

Enqueue

case last != NIL

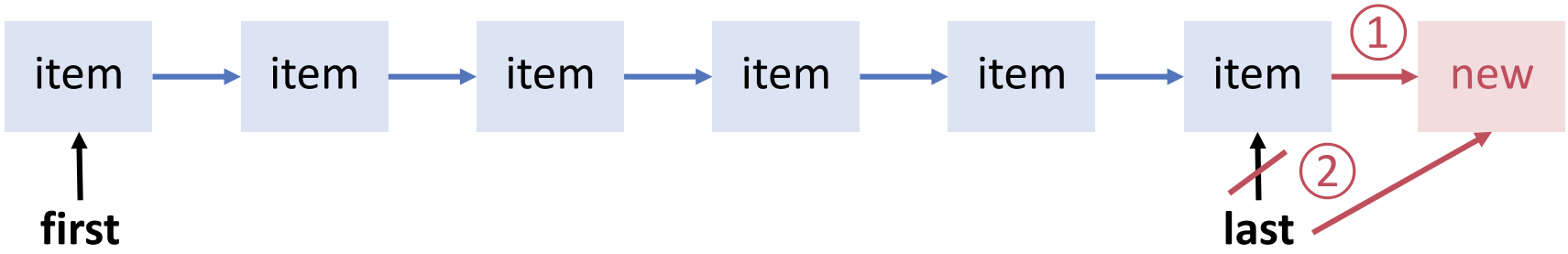


case last = NIL

first **last**

Enqueue

case last != NIL

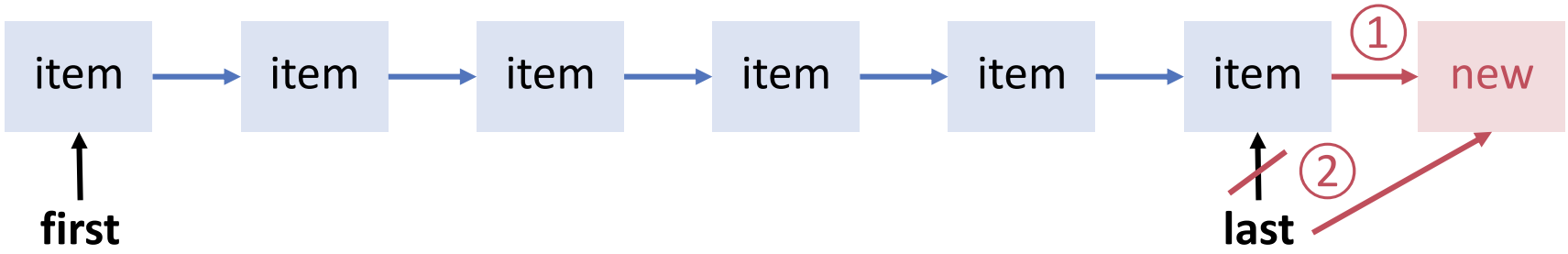


case last = NIL

first **last**

Enqueue

case last != NIL

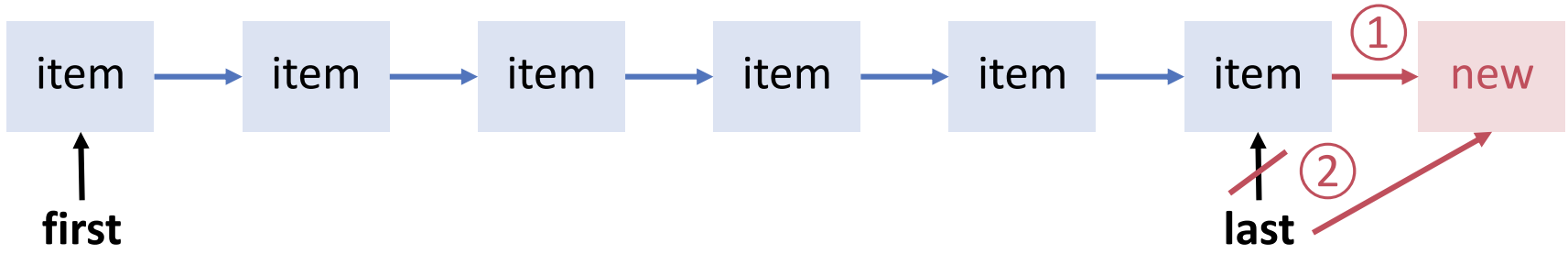


case last = NIL

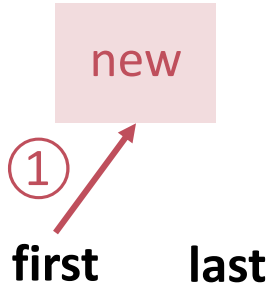
first **last**

Enqueue

case last != NIL

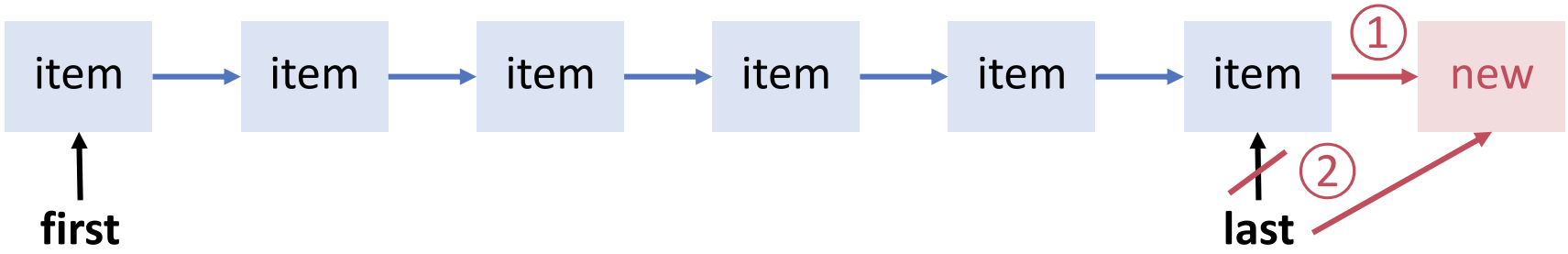


case last = NIL

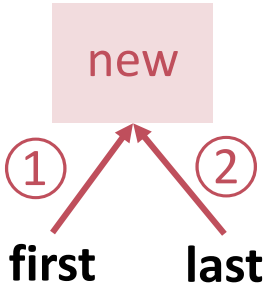


Enqueue

case last != NIL

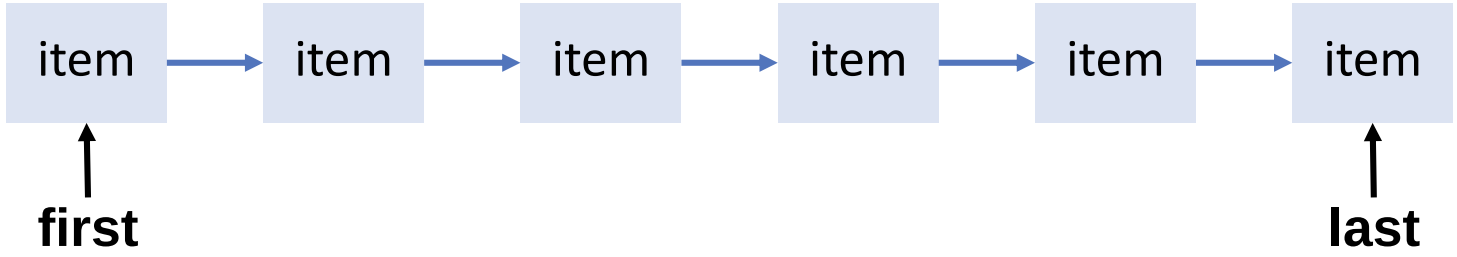


case last = NIL

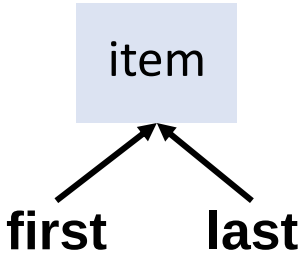


Dequeue

last != first

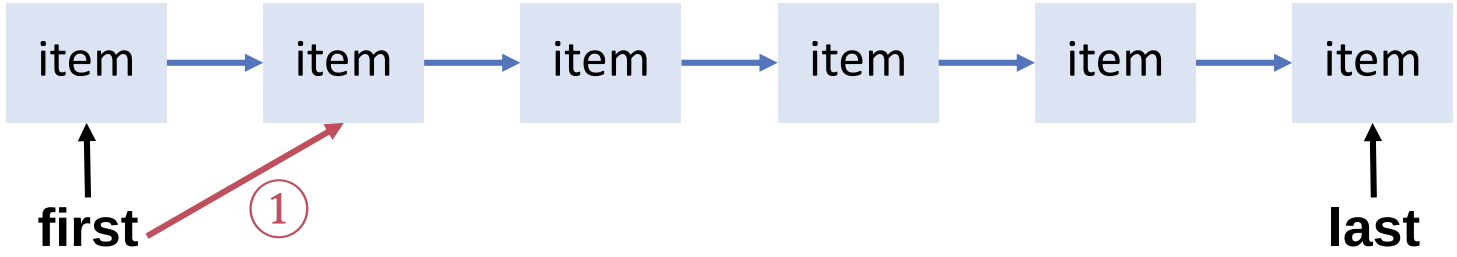


last == first

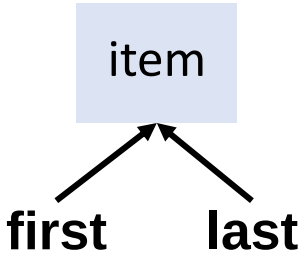


Dequeue

last != first

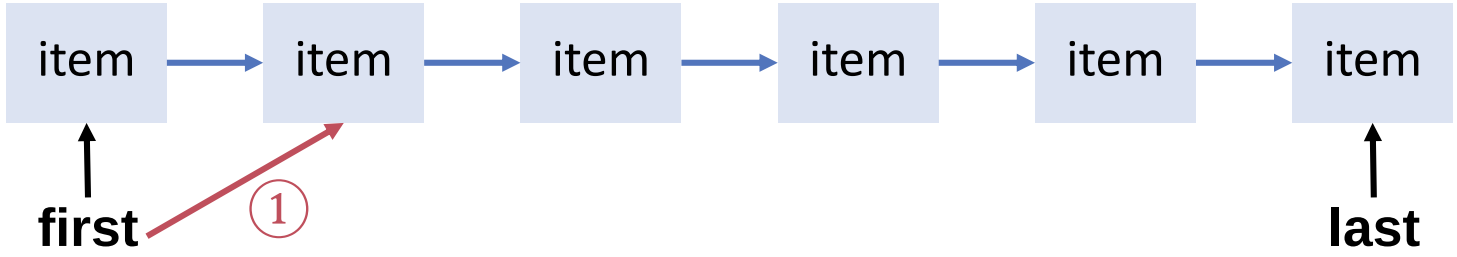


last == first

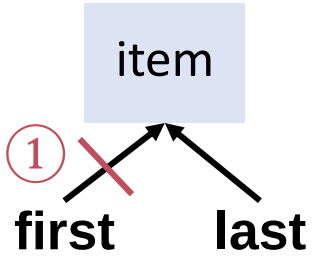


Dequeue

last != first

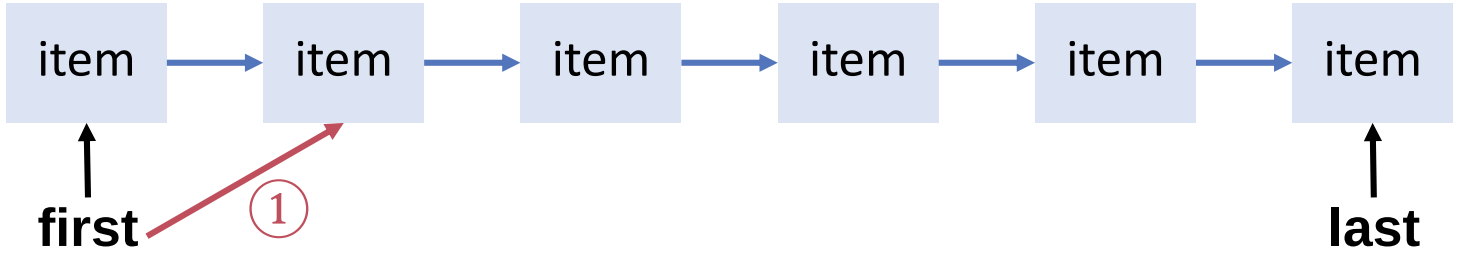


last == first

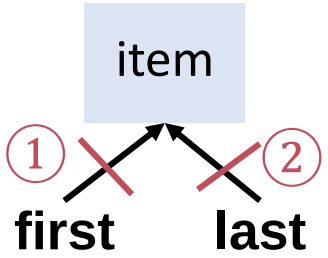


Dequeue

last != first



last == first



Naive Approach

```
Enqueue (q, new)
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
    e2  CAS(last.next, NIL, new);
    ELSE
    e3  CAS(q.first, NIL, new);
    END
```

```
Dequeue (q)
    REPEAT
        first := CAS(q.first, null, null);
    d1  IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
    d3  CAS(q.last, first, NIL);
    END
```

Naive Approach

Enqueue (q, new)

```
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
    e2  CAS(last.next, NIL, new);
    ELSE
    e3  CAS(q.first, NIL, new);
    END
```

Dequeue (q)

```
    REPEAT
        first := CAS(q.first, null, null);
    d1  IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
    d3  CAS(q.last, first, NIL);
    END
```

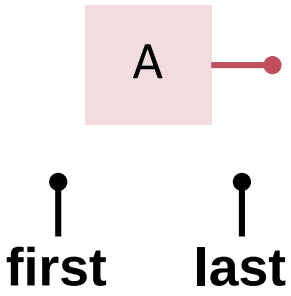
Naive Approach

Enqueue (q, new)

```
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
    e2  CAS(last.next, NIL, new);
    ELSE
    e3  CAS(q.first, NIL, new);
    END
```

Dequeue (q)

```
    REPEAT
        first := CAS(q.first, null, null);
    d1  IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
    d3  CAS(q.last, first, NIL);
    END
```



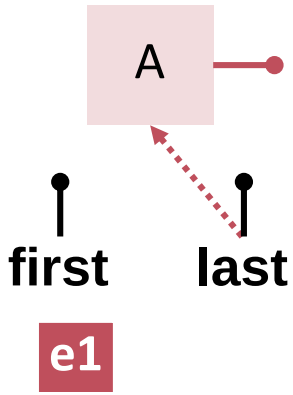
Naive Approach

Enqueue (q, new)

```
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
    e2  CAS(last.next, NIL, new);
    ELSE
    e3  CAS(q.first, NIL, new);
    END
```

Dequeue (q)

```
    REPEAT
        first := CAS(q.first, null, null);
    d1  IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
    d3  CAS(q.last, first, NIL);
    END
```



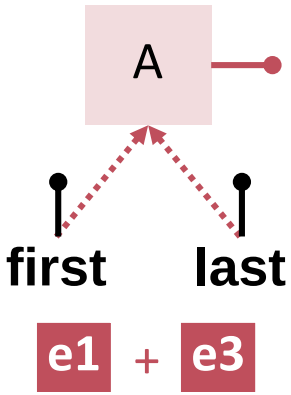
Naive Approach

Enqueue (q, new)

```
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
    e2  CAS(last.next, NIL, new);
    ELSE
    e3  CAS(q.first, NIL, new);
    END
```

Dequeue (q)

```
    REPEAT
        first := CAS(q.first, null, null);
    d1  IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
    d3  CAS(q.last, first, NIL);
    END
```



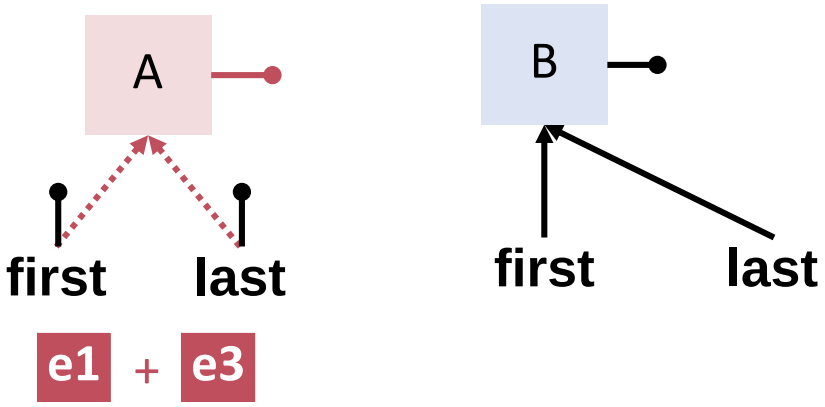
Naive Approach

Enqueue (q, new)

```
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
    e2  CAS(last.next, NIL, new);
    ELSE
    e3  CAS(q.first, NIL, new);
    END
```

Dequeue (q)

```
    REPEAT
        first := CAS(q.first, null, null);
    d1  IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
    d3  CAS(q.last, first, NIL);
    END
```



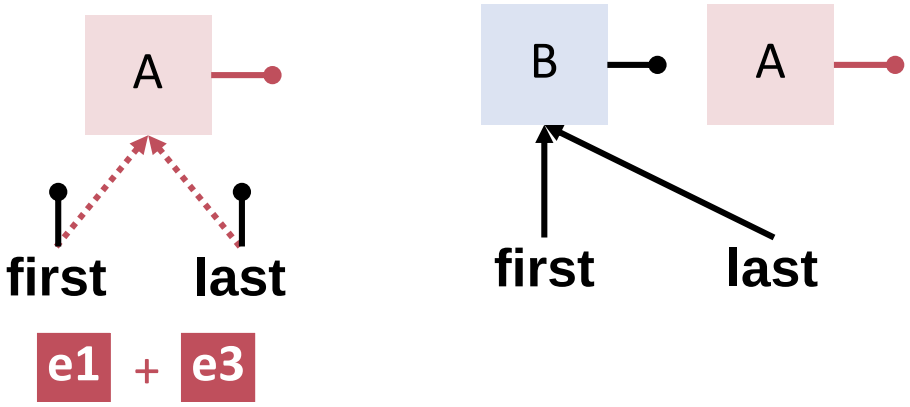
Naive Approach

Enqueue (q, new)

```
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
    e2  CAS(last.next, NIL, new);
    ELSE
    e3  CAS(q.first, NIL, new);
    END
```

Dequeue (q)

```
    REPEAT
        first := CAS(q.first, null, null);
    d1  IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
    d3  CAS(q.last, first, NIL);
    END
```



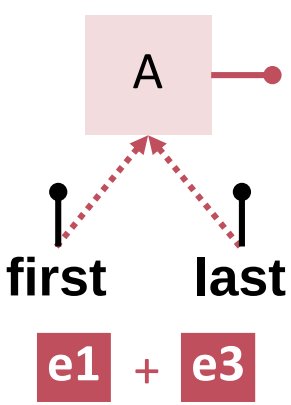
Naive Approach

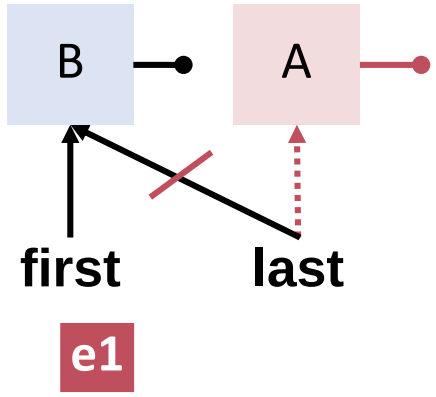
Enqueue (q, new)

```
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
    e2  CAS(last.next, NIL, new);
    ELSE
    e3  CAS(q.first, NIL, new);
    END
```

Dequeue (q)

```
    REPEAT
        first := CAS(q.first, null, null);
    d1  IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
    d3  CAS(q.last, first, NIL);
    END
```





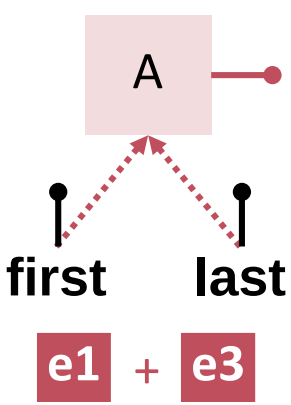
Naive Approach

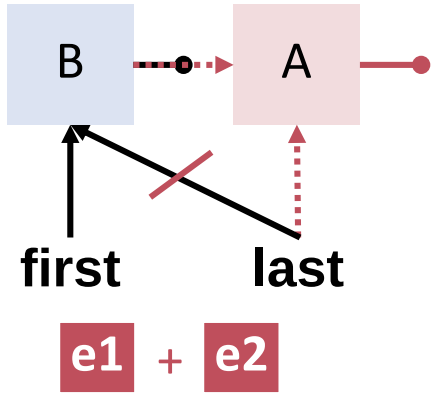
Enqueue (q, new)

```
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
    e2  CAS(last.next, NIL, new);
    ELSE
    e3  CAS(q.first, NIL, new);
    END
```

Dequeue (q)

```
    REPEAT
        first := CAS(q.first, null, null);
    d1  IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
    d3  CAS(q.last, first, NIL);
    END
```





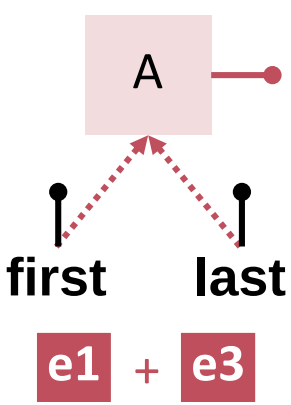
Naive Approach

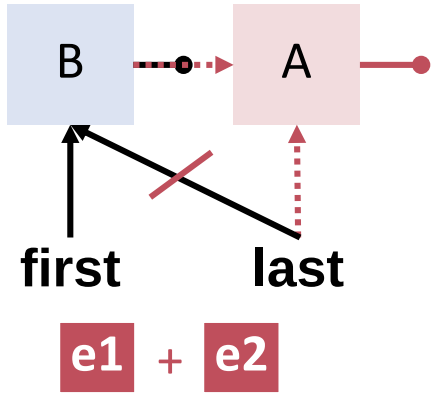
Enqueue (q, new)

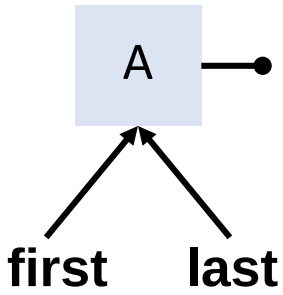
```
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
    e2  CAS(last.next, NIL, new);
    ELSE
    e3  CAS(q.first, NIL, new);
    END
```

Dequeue (q)

```
    REPEAT
        first := CAS(q.first, null, null);
    d1  IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
    d3  CAS(q.last, first, NIL);
    END
```







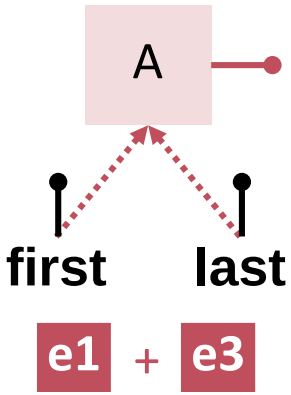
Naive Approach

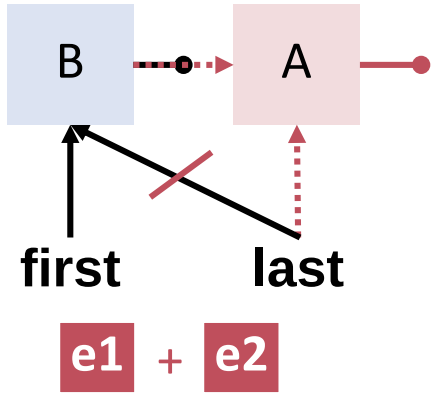
Enqueue (q, new)

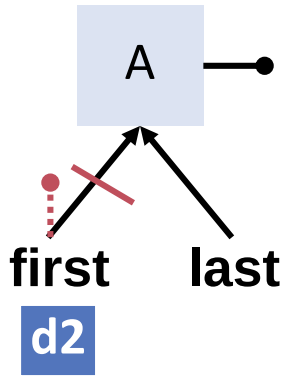
```
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
    e2  CAS(last.next, NIL, new);
    ELSE
    e3  CAS(q.first, NIL, new);
    END
```

Dequeue (q)

```
    REPEAT
        first := CAS(q.first, null, null);
    d1  IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
    d3  CAS(q.last, first, NIL);
    END
```







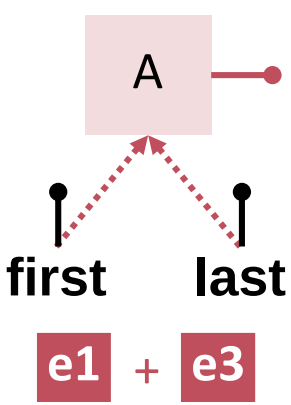
Naive Approach

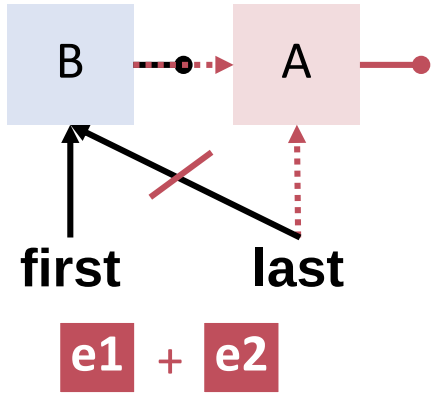
Enqueue (q, new)

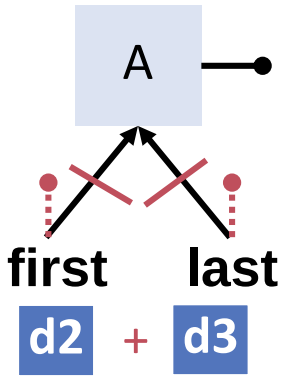
```
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
    e2  CAS(last.next, NIL, new);
    ELSE
    e3  CAS(q.first, NIL, new);
    END
```

Dequeue (q)

```
    REPEAT
        first := CAS(q.first, null, null);
    d1  IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
    d3  CAS(q.last, first, NIL);
    END
```







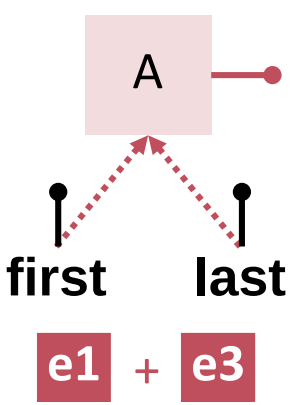
Naive Approach

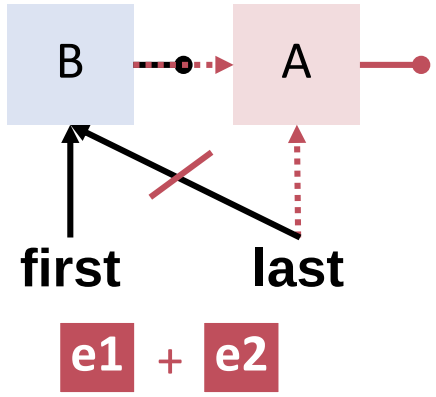
Enqueue (q, new)

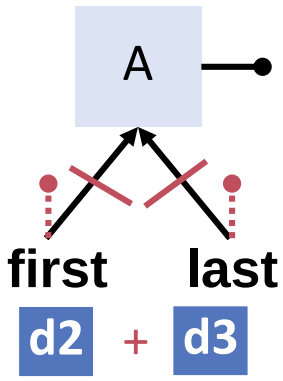
```
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
    e2  CAS(last.next, NIL, new);
    ELSE
    e3  CAS(q.first, NIL, new);
    END
```

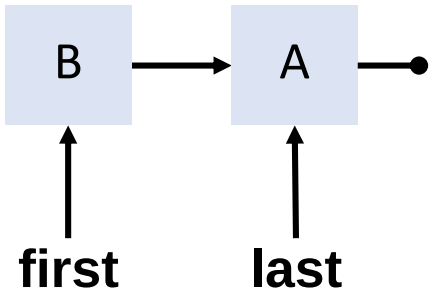
Dequeue (q)

```
    REPEAT
        first := CAS(q.first, null, null);
    d1  IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
    d3  CAS(q.last, first, NIL);
    END
```









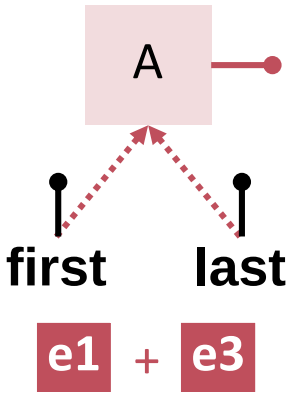
Naive Approach

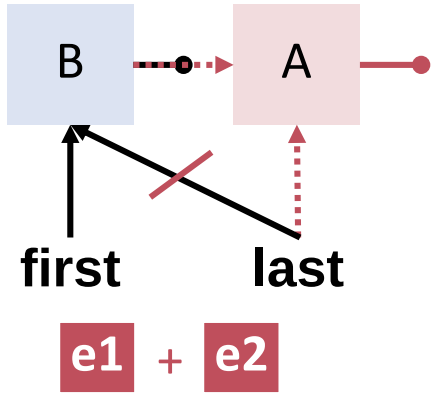
Enqueue (q, new)

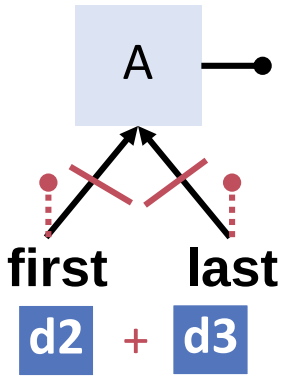
```
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
    e2  CAS(last.next, NIL, new);
    ELSE
    e3  CAS(q.first, NIL, new);
    END
```

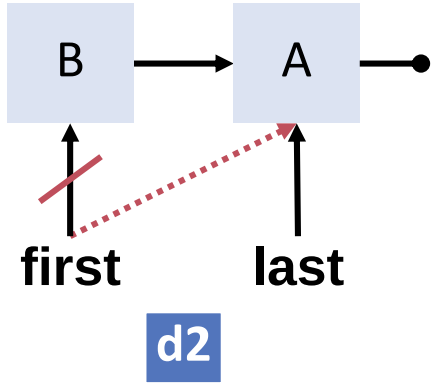
Dequeue (q)

```
    REPEAT
        first := CAS(q.first, null, null);
    d1  IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
    d3  CAS(q.last, first, NIL);
    END
```









Scenario

Process P enqueues A

Process Q dequeues

```
Enqueue (q, new)
      REPEAT last := CAS(q.last, NIL, NIL);
e1    UNTIL CAS(q.last, last, new) = last;
      IF last # NIL THEN
e2          CAS(last.next, NIL, new);
      ELSE
e3          CAS(q.first, NIL, new);
      END
```

```
Dequeue (q)
      REPEAT
            first := CAS(q.first, null, null);
d1          IF first = NIL THEN RETURN NIL END;
            next := first.next;
d2    UNTIL CAS(q.first, first, next) = first;
      IF next = NIL THEN
d3      CAS(q.last, first, NIL);
      END
```

initial

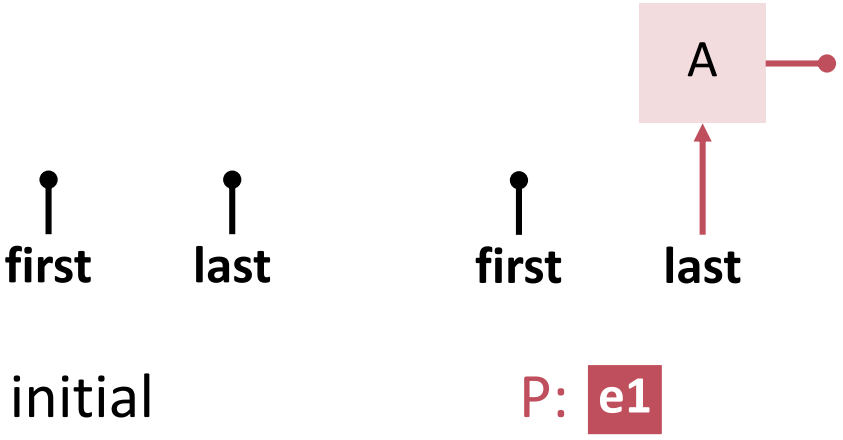
Scenario

Process P enqueues A

Process Q dequeues

```
Enqueue (q, new)
      REPEAT last := CAS(q.last, NIL, NIL);
e1    UNTIL CAS(q.last, last, new) = last;
      IF last # NIL THEN
e2          CAS(last.next, NIL, new);
      ELSE
e3          CAS(q.first, NIL, new);
      END
```

```
Dequeue (q)
      REPEAT
            first := CAS(q.first, null, null);
d1    IF first = NIL THEN RETURN NIL END;
            next := first.next;
d2    UNTIL CAS(q.first, first, next) = first;
      IF next = NIL THEN
d3    CAS(q.last, first, NIL);
      END
```



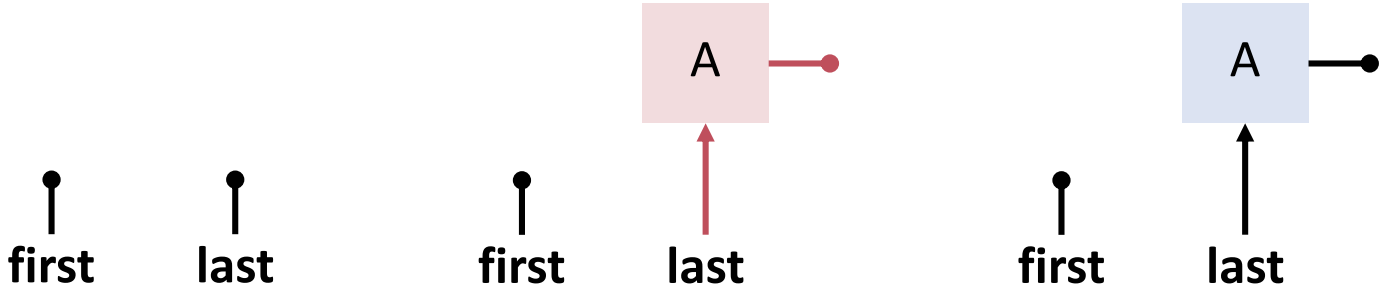
Scenario

Process P enqueues A

Process Q dequeues

```
Enqueue (q, new)
      REPEAT last := CAS(q.last, NIL, NIL);
e1    UNTIL CAS(q.last, last, new) = last;
      IF last # NIL THEN
e2          CAS(last.next, NIL, new);
      ELSE
e3          CAS(q.first, NIL, new);
      END
```

```
Dequeue (q)
      REPEAT
            first := CAS(q.first, null, null);
d1    IF first = NIL THEN RETURN NIL END;
            next := first.next;
d2    UNTIL CAS(q.first, first, next) = first;
      IF next = NIL THEN
d3    CAS(q.last, first, NIL);
      END
```



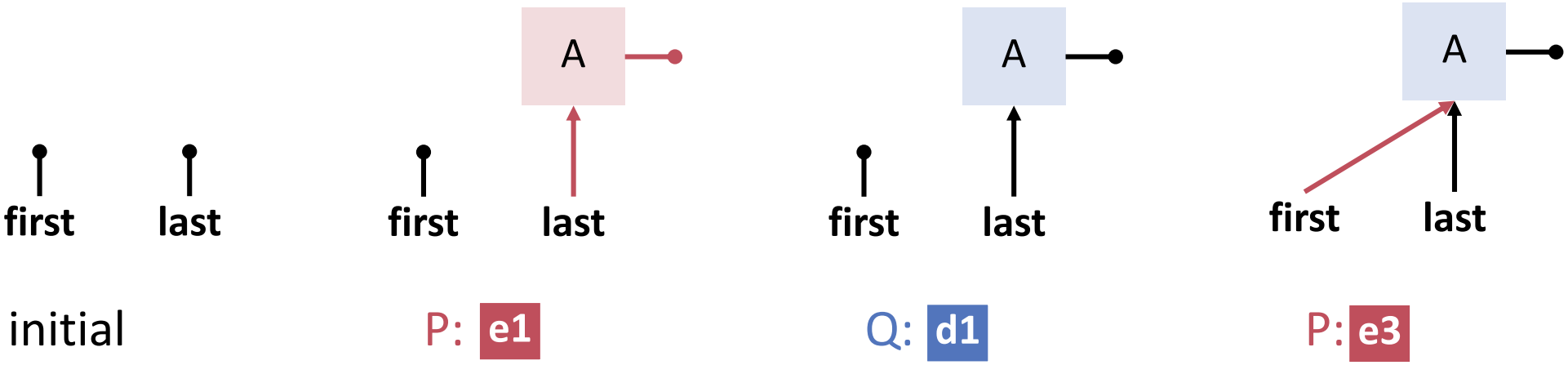
Scenario

Process P enqueues A

Process Q dequeues

```
Enqueue (q, new)
     REPEAT last := CAS(q.last, NIL, NIL);
e1   UNTIL CAS(q.last, last, new) = last;
     IF last # NIL THEN
e2        CAS(last.next, NIL, new);
     ELSE
e3        CAS(q.first, NIL, new);
     END
```

```
Dequeue (q)
     REPEAT
          first := CAS(q.first, null, null);
d1        IF first = NIL THEN RETURN NIL END;
          next := first.next;
d2   UNTIL CAS(q.first, first, next) = first;
     IF next = NIL THEN
d3        CAS(q.last, first, NIL);
     END
```



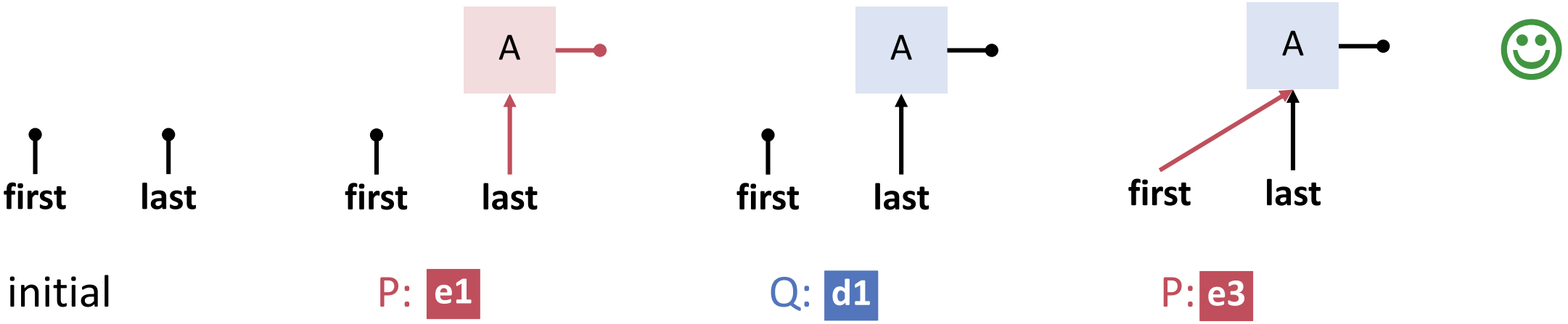
Scenario

Process P enqueues A

Process Q dequeues

```
Enqueue (q, new)
      REPEAT last := CAS(q.last, NIL, NIL);
e1    UNTIL CAS(q.last, last, new) = last;
      IF last # NIL THEN
e2          CAS(last.next, NIL, new);
      ELSE
e3          CAS(q.first, NIL, new);
      END
```

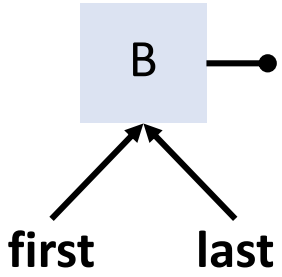
```
Dequeue (q)
      REPEAT
          first := CAS(q.first, null, null);
d1        IF first = NIL THEN RETURN NIL END;
          next := first.next;
d2    UNTIL CAS(q.first, first, next) = first;
      IF next = NIL THEN
d3        CAS(q.last, first, NIL);
      END
```



Scenario

Process P enqueues A

Process Q dequeues



initial

```
Enqueue (q, new)
      REPEAT last := CAS(q.last, NIL, NIL);
e1    UNTIL CAS(q.last, last, new) = last;
      IF last # NIL THEN
e2          CAS(last.next, NIL, new);
      ELSE
e3          CAS(q.first, NIL, new);
      END
```

```
Dequeue (q)
      REPEAT
            first := CAS(q.first, null, null);
d1          IF first = NIL THEN RETURN NIL END;
            next := first.next;
d2    UNTIL CAS(q.first, first, next) = first;
      IF next = NIL THEN
d3      CAS(q.last, first, NIL);
      END
```

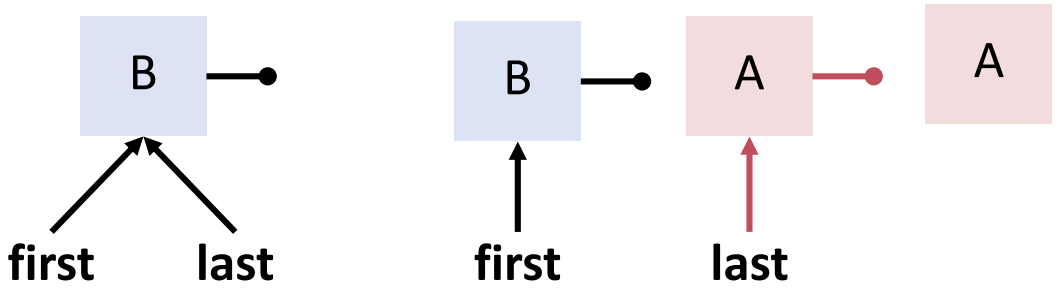
Scenario

Process P enqueues A

Process Q dequeues

```
Enqueue (q, new)
      REPEAT last := CAS(q.last, NIL, NIL);
e1    UNTIL CAS(q.last, last, new) = last;
      IF last # NIL THEN
e2          CAS(last.next, NIL, new);
      ELSE
e3          CAS(q.first, NIL, new);
      END
```

```
Dequeue (q)
      REPEAT
            first := CAS(q.first, null, null);
d1    IF first = NIL THEN RETURN NIL END;
            next := first.next;
d2    UNTIL CAS(q.first, first, next) = first;
      IF next = NIL THEN
d3    CAS(q.last, first, NIL);
      END
```



initial

P: e1

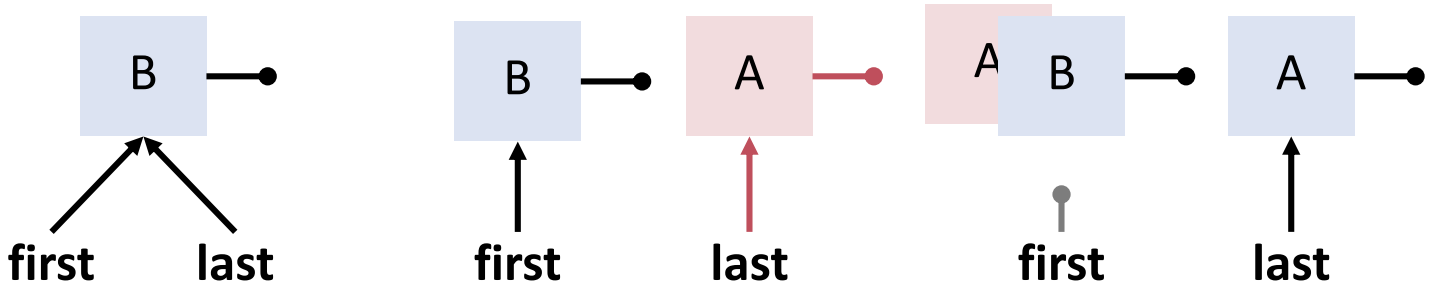
Scenario

Process P enqueues A

Process Q dequeues

```
Enqueue (q, new)
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
e2      CAS(last.next, NIL, new);
    ELSE
e3      CAS(q.first, NIL, new);
    END
```

```
Dequeue (q)
    REPEAT
        first := CAS(q.first, null, null);
d1      IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
d3      CAS(q.last, first, NIL);
    END
```



initial P: e1 Q: d2

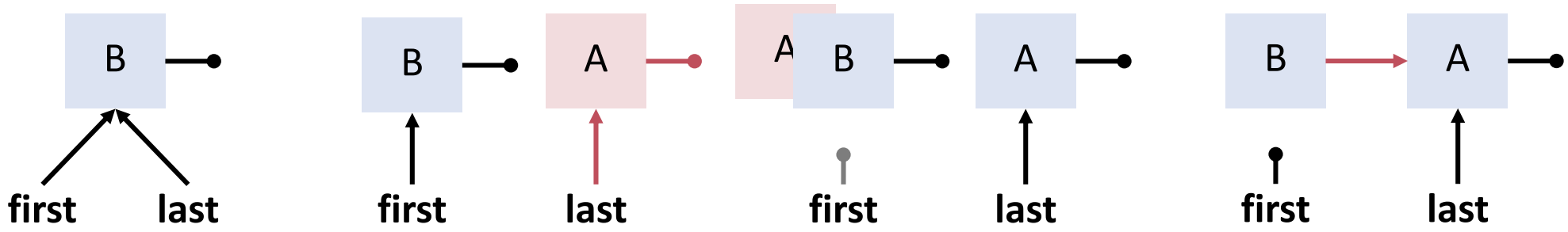
Scenario

Process P enqueues A

Process Q dequeues

```
Enqueue (q, new)
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
e2      CAS(last.next, NIL, new);
    ELSE
e3      CAS(q.first, NIL, new);
    END
```

```
Dequeue (q)
    REPEAT
        first := CAS(q.first, null, null);
d1      IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
d3      CAS(q.last, first, NIL);
    END
```



initial P: e1 Q: d2 P:

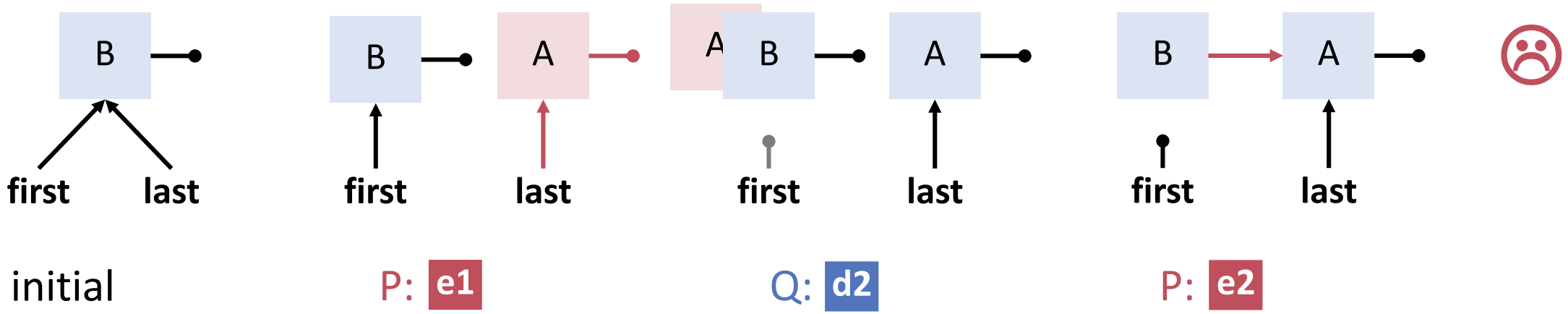
Scenario

Process P enqueues A

Process Q dequeues

```
Enqueue (q, new)
    REPEAT last := CAS(q.last, NIL, NIL);
e1  UNTIL CAS(q.last, last, new) = last;
    IF last # NIL THEN
e2      CAS(last.next, NIL, new);
    ELSE
e3      CAS(q.first, NIL, new);
    END
```

```
Dequeue (q)
    REPEAT
        first := CAS(q.first, null, null);
d1      IF first = NIL THEN RETURN NIL END;
        next := first.next;
d2  UNTIL CAS(q.first, first, next) = first;
    IF next = NIL THEN
d3      CAS(q.last, first, NIL);
    END
```



Analysis

Analysis

- The problem is that enqueue and dequeue do under some circumstances have to update **several pointers at once** [first, last, next]

Analysis

- The problem is that enqueue and dequeue do under some circumstances have to update **several pointers at once** [first, last, next]
- The transient inconsistency can lead to permanent data structure corruption

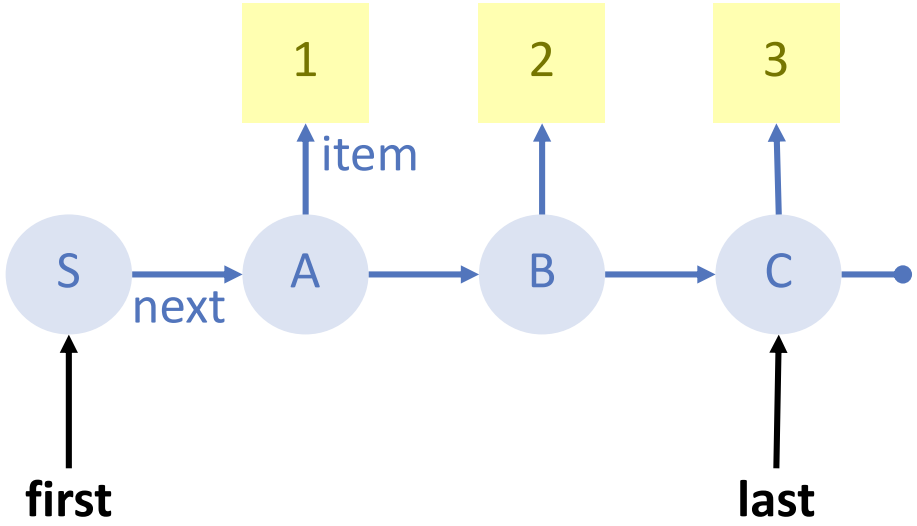
Analysis

- The problem is that enqueue and dequeue do under some circumstances have to update **several pointers at once** [first, last, next]
- The transient inconsistency can lead to permanent data structure corruption
- Solutions to this particular problem are not easy to find if no double compare and swap (or similar) is available

Analysis

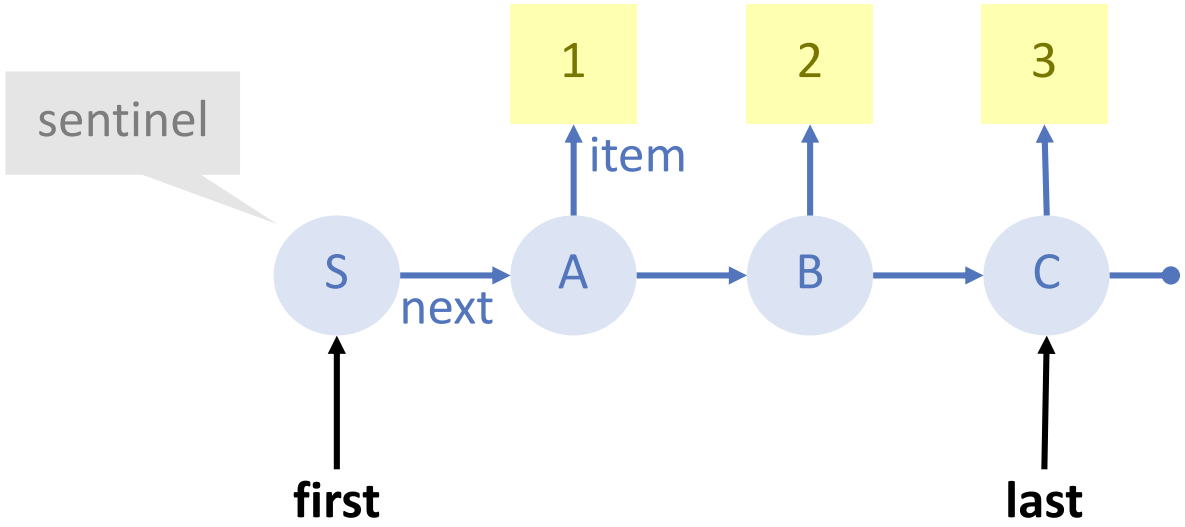
- The problem is that enqueue and dequeue do under some circumstances have to update **several pointers at once** [first, last, next]
- The transient inconsistency can lead to permanent data structure corruption
- Solutions to this particular problem are not easy to find if no double compare and swap (or similar) is available
- Need another approach: Decouple enqueue and dequeue with a sentinel. A consequence is that the **queue cannot be in-place.**

Queues with Sentinel



Queue empty:	first = last
Queue nonempty:	first # last
Invariants:	first # NIL
	last # NIL

Queues with Sentinel



Queue empty: first = last

Queue nonempty: first # last

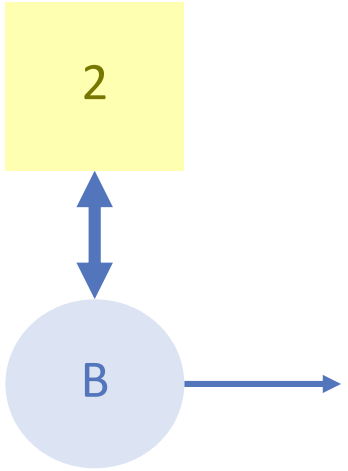
Invariants: first # NIL

last # NIL

Node Reuse

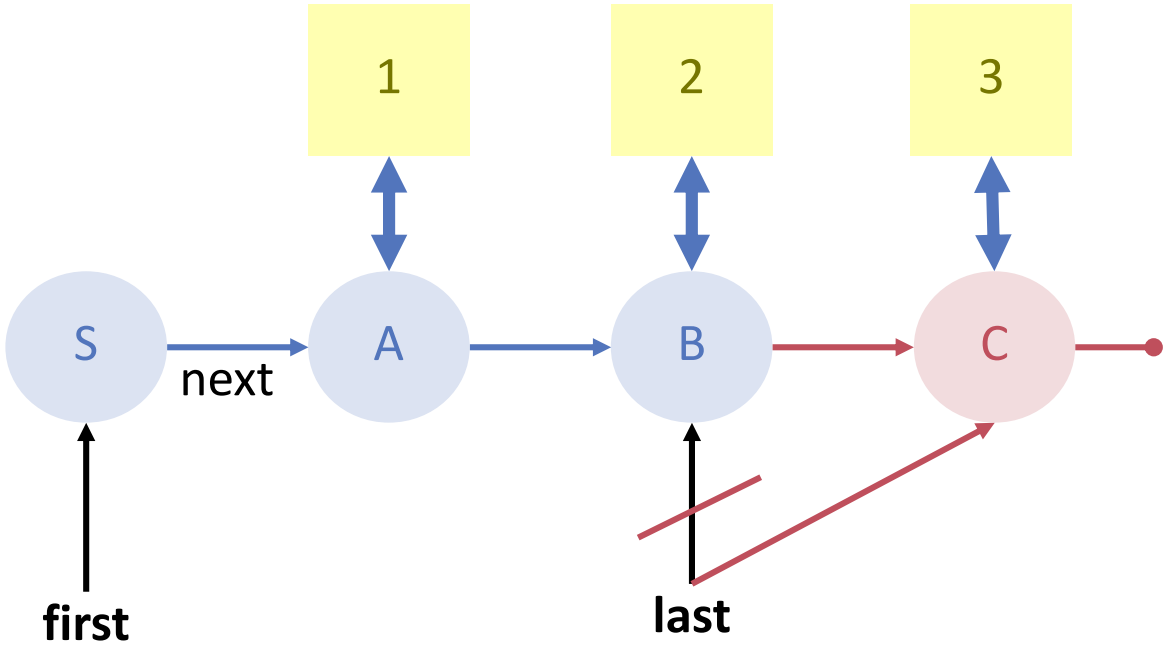
simple idea:

link from node to item
and from item to node

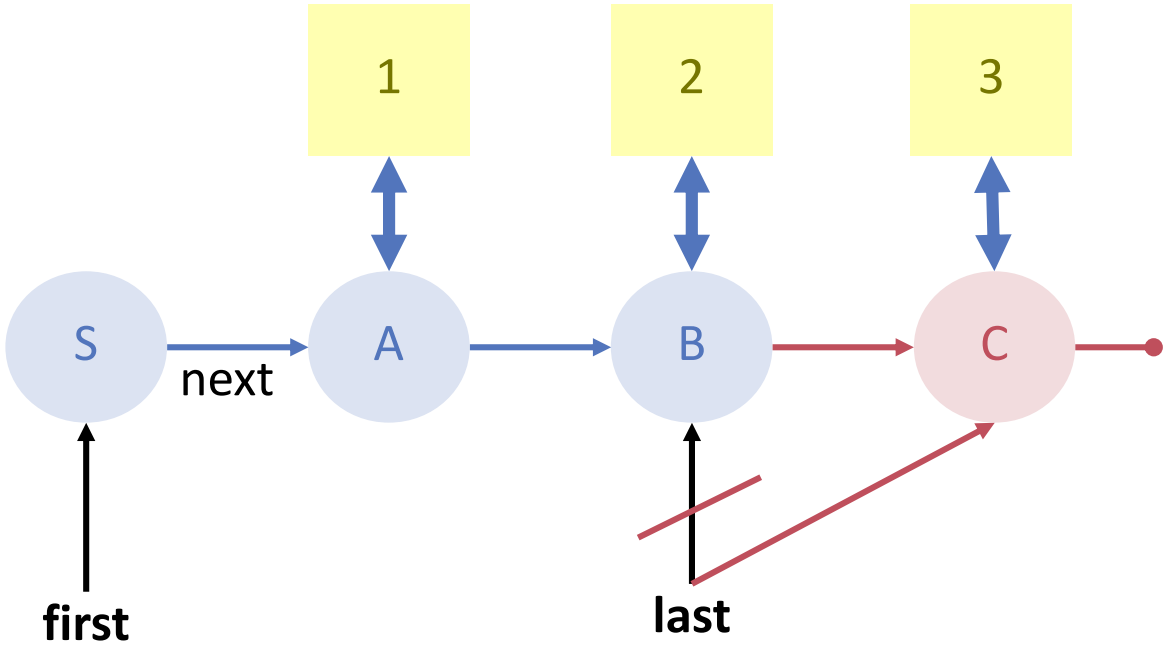


Enqueue and Dequeue with Sentinel

Enqueue and Dequeue with Sentinel

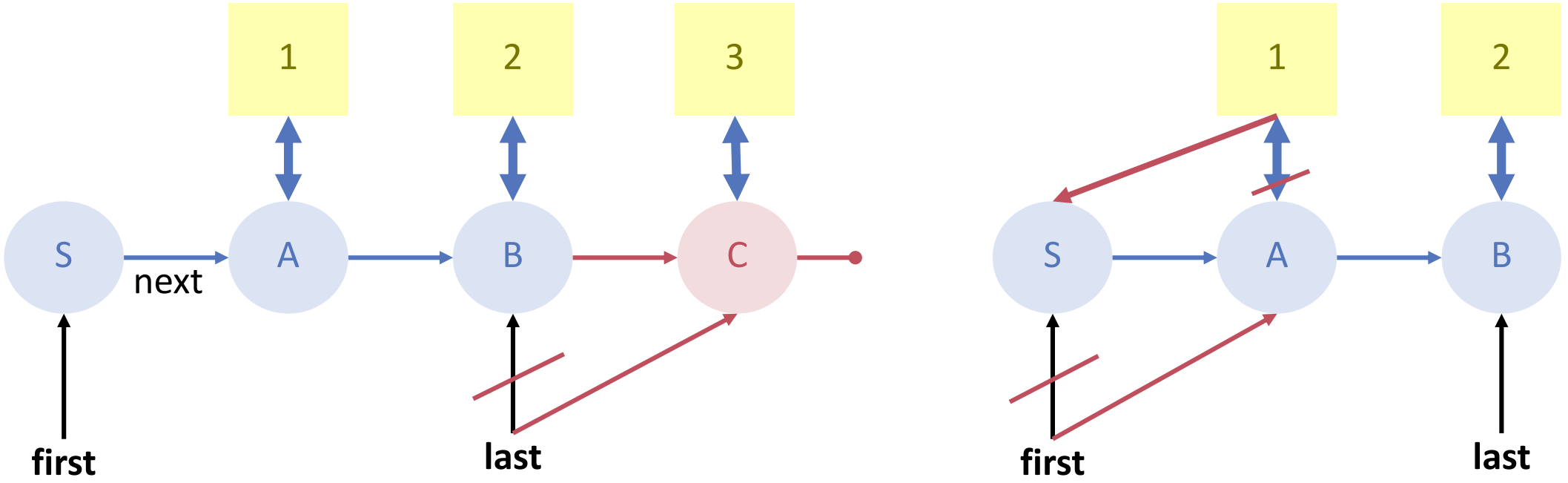


Enqueue and Dequeue with Sentinel



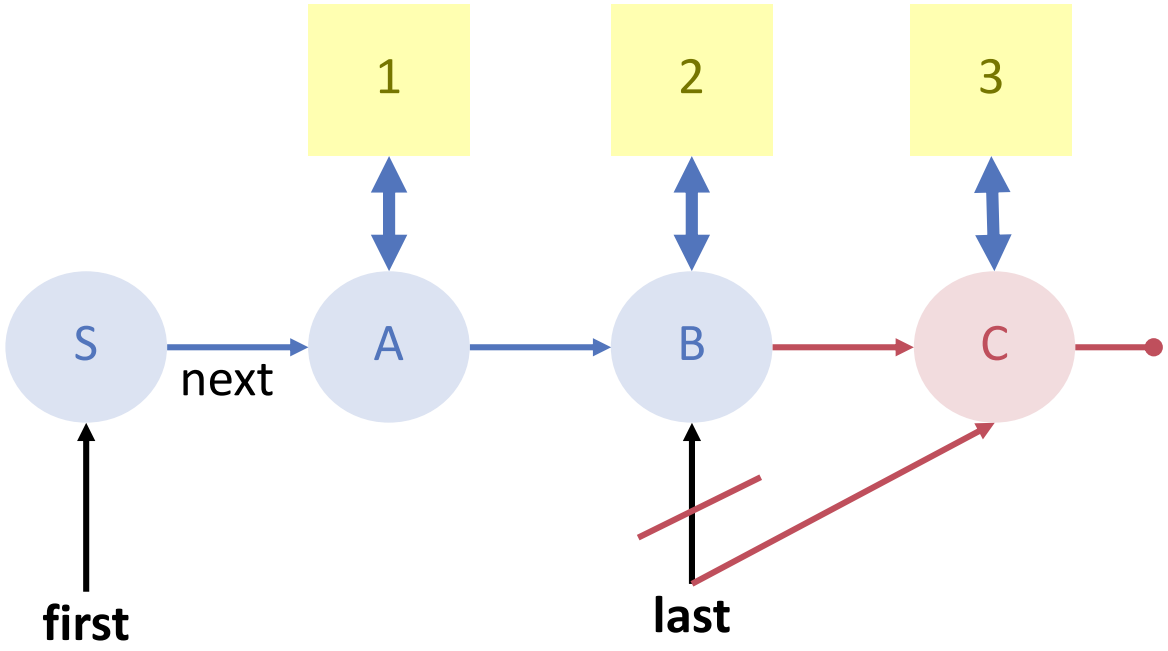
Item enqueued together with associated node.

Enqueue and Dequeue with Sentinel

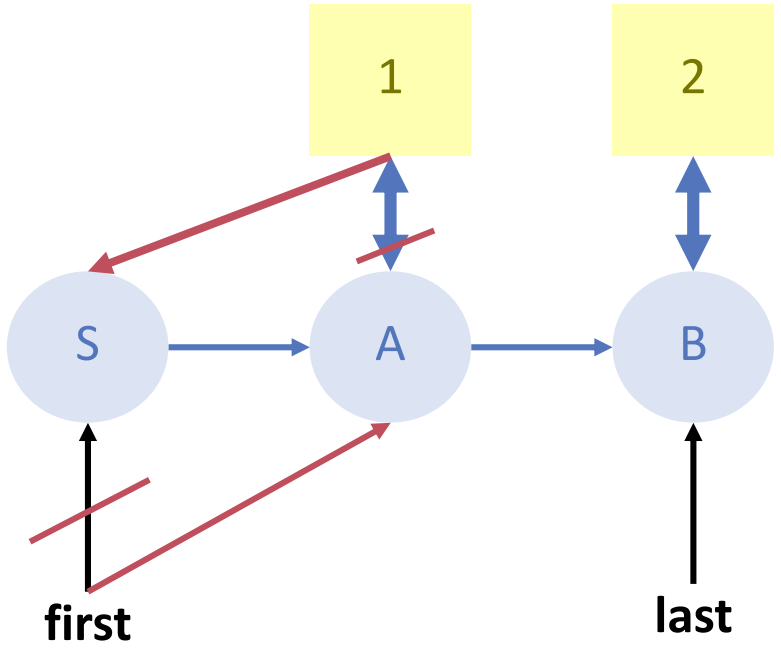


Item enqueued together with associated node.

Enqueue and Dequeue with Sentinel



Item enqueued together with associated node.



A becomes the new sentinel.
S associated with free item.

Enqueue

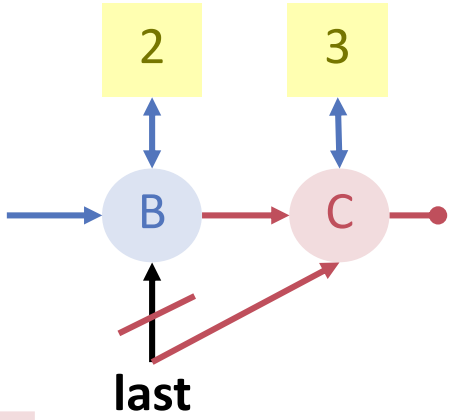
```
PROCEDURE Enqueue- (item: Item; VAR queue: Queue);
VAR node, last, next: Node;
BEGIN
   node := Allocate();
   node.item := Item:
   LOOP
      last := CAS (queue.last, NIL, NIL);
      next := CAS (last.next, NIL, node);
      IF next = NIL THEN EXIT END;
      IF CAS (queue.last, last, next) # last THEN CPU.Backoff END;
   END;
   ASSERT (CAS (queue.last, last, node) # NIL);
END Enqueue;
```

Set last node's next pointer

If setting last pointer failed, then **help other processes to update last node → Progress guarantee**

Set last node, can fail but then others have already helped

Enqueue




```
PROCEDURE Enqueue- (item: Item; VAR queue: Queue);
VAR node, last, next: Node;
BEGIN
   node := Allocate();
   node.item := Item:
   LOOP
      last := CAS (queue.last, NIL, NIL);
      next := CAS (last.next, NIL, node);
      IF next = NIL THEN EXIT END;
      IF CAS (queue.last, last, next) # last THEN CPU.Backoff END;
   END;
   ASSERT (CAS (queue.last, last, node) # NIL);
END Enqueue;
```

Set last node's next pointer

If setting last pointer failed, then **help other processes to update last node → Progress guarantee**

Set last node, can fail but then others have already helped

Dequeue

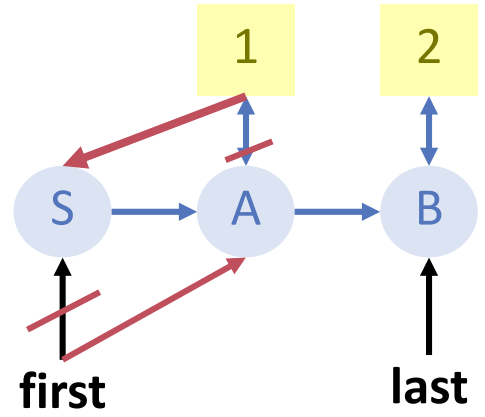
```
PROCEDURE Dequeue- (VAR item: Item; VAR queue: Queue): BOOLEAN;
VAR first, next, last: Node;
BEGIN
   LOOP
      first := CAS (queue.first, NIL, NIL);
      next := CAS (first.next, NIL, NIL);
      IF next = NIL THEN RETURN FALSE END;
      last := CAS (queue.last, first, next);
      item := next.item;
      IF CAS (queue.first, first, next) = first THEN EXIT END;
      CPU.Backoff;
   END;
   item.node := first;
   RETURN TRUE;
END Dequeue;
```

Remove potential inconsistency, **help other processes to set last pointer**

set first pointer

associate node with first

Dequeue




```
PROCEDURE Dequeue- (VAR item: Item; VAR queue: Queue): BOOLEAN;
VAR first, next, last: Node;
BEGIN
   LOOP
      first := CAS (queue.first, NIL, NIL);
      next := CAS (first.next, NIL, NIL);
      IF next = NIL THEN RETURN FALSE END;
      last := CAS (queue.last, first, next);
      item := next.item;
      IF CAS (queue.first, first, next) = first THEN EXIT END;
      CPU.Backoff;
   END;
   item.node := first;
   RETURN TRUE;
END Dequeue;
```

Remove potential inconsistency, **help other processes to set last pointer**

set first pointer

associate node with first