Local Reasoning for RCU

Hongseok Yang (Queen Mary, Univ. of London)

Joint work with
Peter O'Hearn, Matthew Parkinson, Noam Rinetzky, Mooly Sagiv
and Viktor Vafeiadis

Preparing Multicore Era

- Active research by verification communities.
- Various workshops/conferences (E.g. EC^2).
- At least 9 CAV papers on concurrency.
- Concurrency is a hot topic inside separation logic.

Preparing Multicore Era

- Active research by verification communities.
- Various workshops/conferences (E.g. EC^2).
- At least 9 CAV papers on concurrency.
- Concurrency is a hot topic inside separation logic.
- What about Linux hackers?

We have demonstrated that RGSep and shape-value abstraction enable effective automatic linearizability proofs. The examples verified are typical of the research literature 5–10 years ago. The techniques can also cope with more complex al-

From "Shape-value abstraction for verifying linearizability" by Viktor Vafeiadis

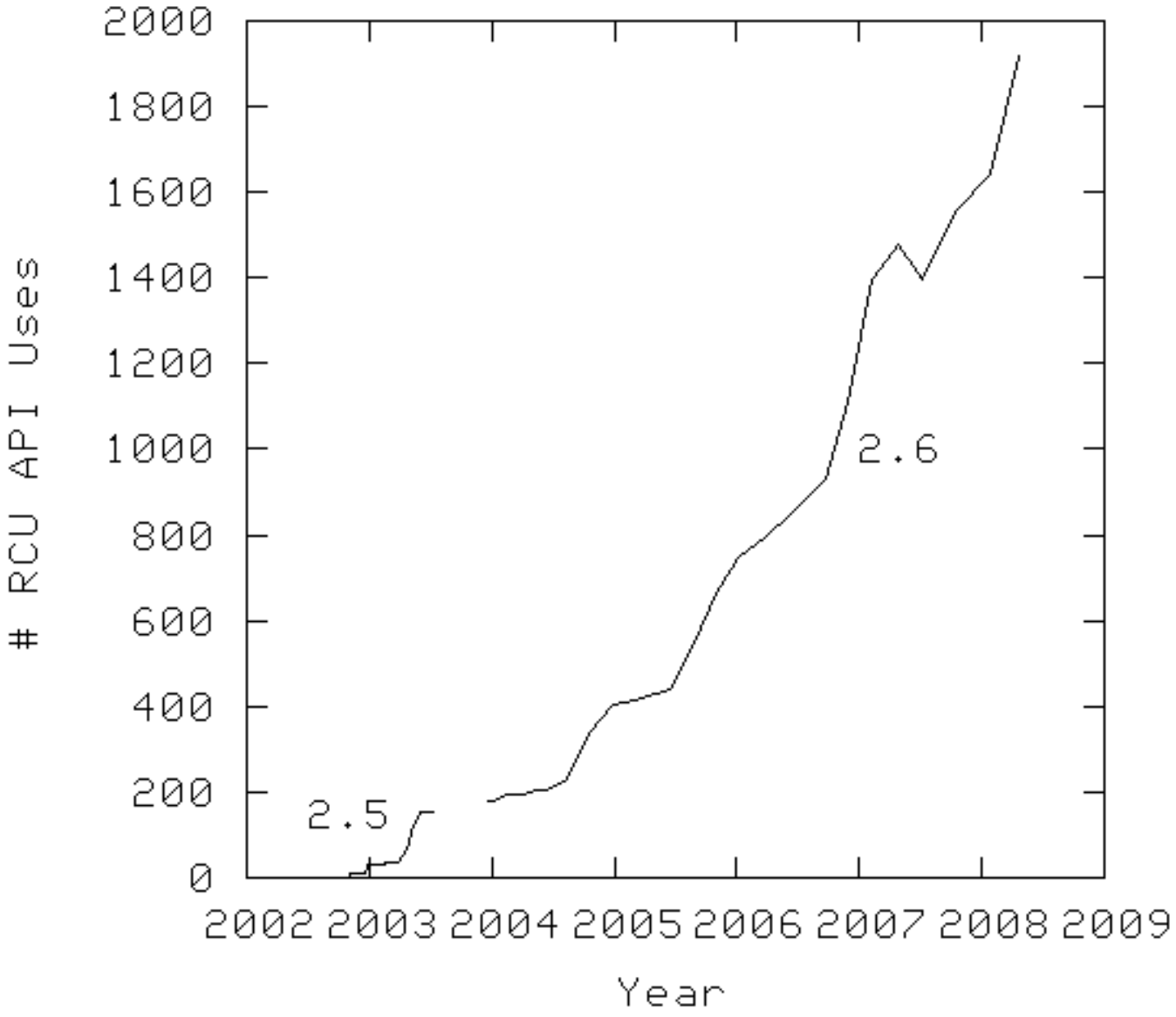
- Active research by verification communities.
- Various workshops/conferences (E.g. EC^2).
- At least 9 CAV papers on concurrency.
- Concurrency is a hot topic inside separation logic.
- What about Linux hackers?

We have demonstrated that RGSep and shape-value abstraction enable effective automatic linearizability proofs. The examples verified are typical of the research literature 5–10 years ago. The techniques can also cope with more complex al-

From "Shape-value abstraction for verifying linearizability" by Viktor Vafeiadis

- Active research by verification communities.
- Various workshops/conferences (E.g. EC^2).
- At least 9 CAV papers on concurrency.
- Concurrency is a hot topic inside separation logic.
- What about Linux hackers?

RCU Usage in Linux



Data from
Paul McKenney's
RCU Web Page

Objectives

- Explain RCU and research problems related to RCU.
- Present a preliminary result on program logic for RCU.

RCU (Read-Copy-Update)

- Back to the future: old style locking for new style architecture.
- Very approximately, reader/writer locks.
- Allows the concurrent access by multiple readers and a single writer.
- Almost zero concurrency overhead for readers.
- Intended to work well with weak memory.

RCU Reader/ Writer

```
rcu_read_lock {
    b' = b;
    x = *b';
    y = *(b'+1);
}
.. USE x,y ..
```

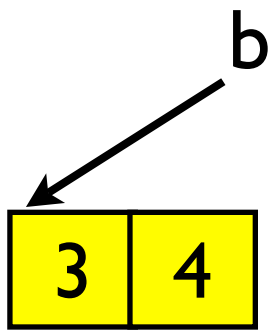
```
.. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;
    barrier();
    ob = b; b = nb;
} sync {
   free(ob); free(ob+1); }
```

reader0

reader1

writer0

writer1



RCU Reader/ Writer

```
rcu_read_lock {
    b' = b;
    x = *b';
    y = *(b'+1);
}
.. USE x,y ..
```

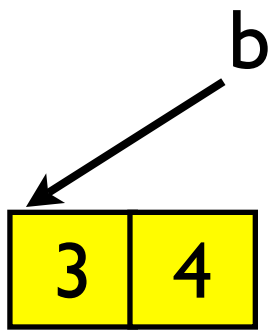
```
.. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;
    barrier();
    ob = b; b = nb;
} sync {
   free(ob); free(ob+1); }
```

reader0

reader1

writer0

writer1



```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);

*nb = u; *(nb+1) = v;

ob = b; b = nb;

free(ob); free(ob+1);
```

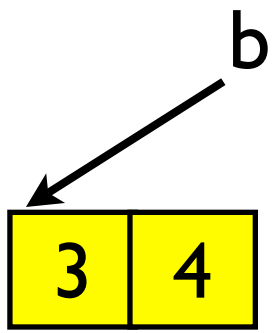
RCU Reader/ Writer

reader0

reader1

writer0

writer1



RCU Reader/ Writer

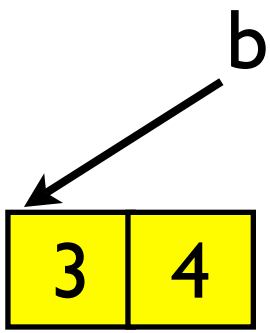
```
    b' = b;
    x = *b';
    y = *(b'+1);


.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);

   *nb = u; *(nb+1) = v;

   ob = b; b = nb;

  free(ob); free(ob+1);
```

reader0

x: , y:
b'

reader1



writer0

writer1

```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);

*nb = u; *(nb+1) = v;

ob = b; b = nb;

free(ob); free(ob+1);
```

RCU Reader/ Writer

reader0

x: , y:

b'

b

3 | 4

writer0

reader1

writer1

```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);

*nb = u; *(nb+1) = v;

ob = b; b = nb;

free(ob); free(ob+1);
```

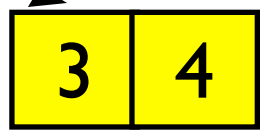
RCU Reader/ Writer

reader0

x:3 , y:

b'

b



writer0

reader1

writer1

```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);

*nb = u; *(nb+1) = v;

ob = b; b = nb;

free(ob); free(ob+1);
```

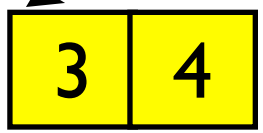
RCU Reader/ Writer

reader0

x:3 , y:4

b'

b



writer0

reader1

writer1

```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);

   *nb = u; *(nb+1) = v;

   ob = b; b = nb;

  free(ob); free(ob+1);
```

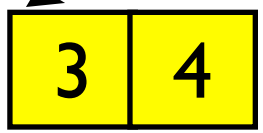
RCU Reader/ Writer

reader0

x:3 , y:4

b'

b



writer0

reader1

writer1

```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);

*nb = u; *(nb+1) = v;

ob = b; b = nb;

free(ob); free(ob+1);
```

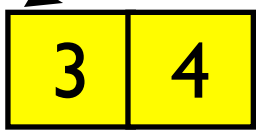
RCU Reader/ Writer

reader0

x:3 , y:4

b'

b



writer0

u:11, v:12

ob nb

reader1

writer1

```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);

*nb = u; *(nb+1) = v;

ob = b; b = nb;

free(ob); free(ob+1);
```

RCU Reader/ Writer

reader0

x:3 , y:4

b'

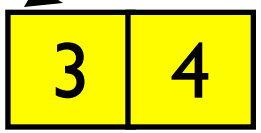
reader1

writer0

u:11, v:12

ob nb

b



writer1

```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);

*nb = u; *(nb+1) = v;

ob = b; b = nb;

free(ob); free(ob+1);
```

RCU Reader/ Writer

reader0

x:3 , y:4

b'

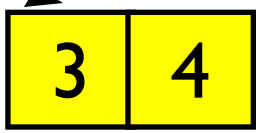
reader1

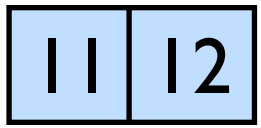
writer0

u:11, v:12

ob nb

b





writer1

```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);

*nb = u; *(nb+1) = v;

ob = b; b = nb;

free(ob); free(ob+1);
```

RCU Reader/ Writer

reader0

x:3 , y:4

b'

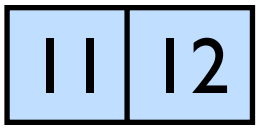
reader1

writer0

u:11, v:12

ob nb

b



writer1

```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);

*nb = u; *(nb+1) = v;

ob = b; b = nb;

free(ob); free(ob+1);
```

RCU Reader/ Writer

reader0

x:3 , y:4

b'

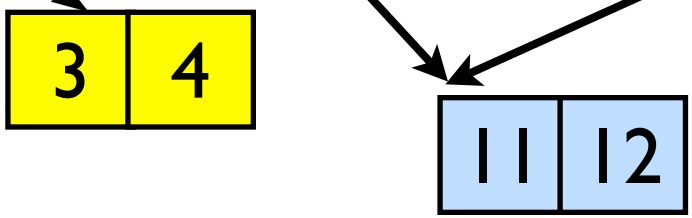
reader1

writer0

u:11, v:12

ob nb

b



writer1

```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);

*nb = u; *(nb+1) = v;

ob = b; b = nb;

free(ob); free(ob+1);
```

RCU Reader/ Writer

reader0

x:3 , y:4

b'

reader1

writer0

u:11, v:12

ob nb

b

11 12

writer1

```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);

*nb = u; *(nb+1) = v;

ob = b; b = nb;

free(ob); free(ob+1);
```

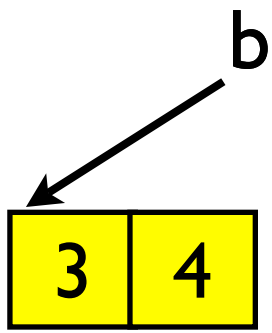
RCU Reader/ Writer

reader0

reader1

writer0

writer1



```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);

*nb = u; *(nb+1) = v;

ob = b; b = nb;

free(ob); free(ob+1);
```

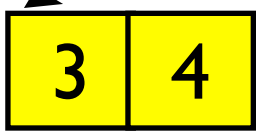
RCU Reader/ Writer

reader0

Pb1: Racy writers.

writer0

b



reader1

writer1

```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

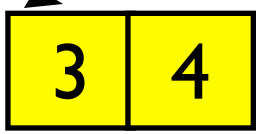
```
.. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;

    ob = b; b = nb;

    free(ob); free(ob+1); }
```

RCU Reader/ Writer

reader0

Pb1: Racy writers.
Sol: Use a standard lock.

b



writer0

reader1

writer1

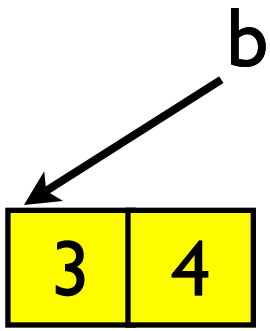
RCU Reader/ Writer

```
    b' = b;
    x = *b';
    y = *(b'+1);


.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;


    ob = b; b = nb;


   free(ob); free(ob+1); }
```

reader0

writer0



reader1

writer1

Pb2: Dangling pointer dereference by readers.

```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;

    ob = b; b = nb;

    free(ob); free(ob+1); }
```

RCU Reader/ Writer

reader0

writer0

u:11, v:12

ob nb

b

3 4

11 12

reader1

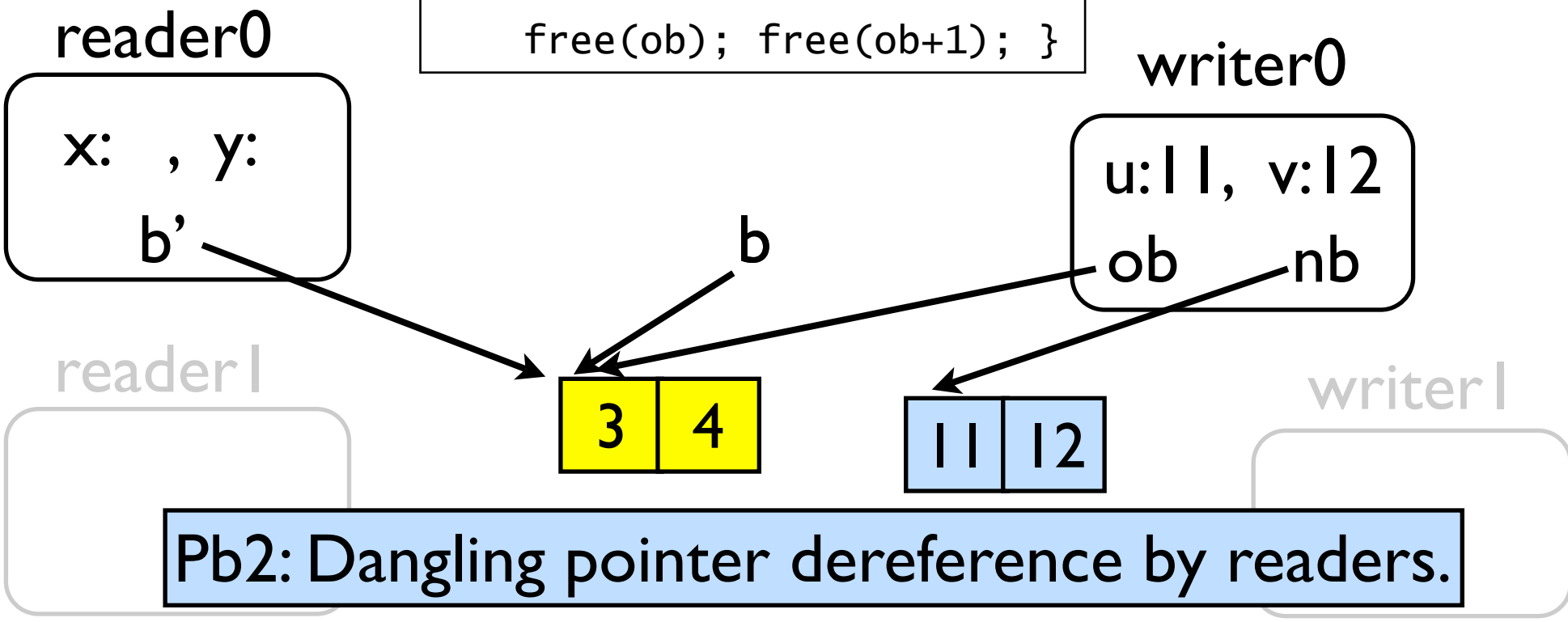
writer1

Pb2: Dangling pointer dereference by readers.

```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;

    ob = b; b = nb;

  free(ob); free(ob+1); }
```

RCU Reader/ Writer

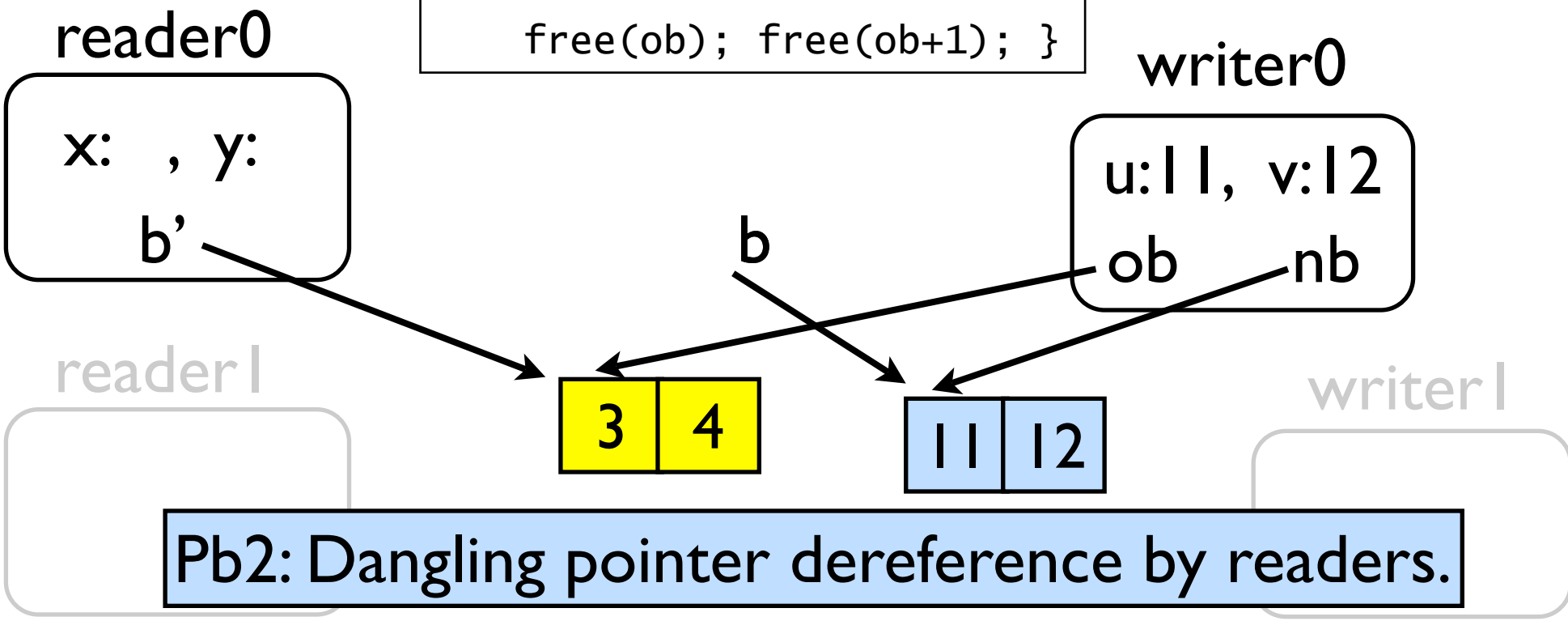


Pb2: Dangling pointer dereference by readers.

```
b' = b;
x = *b';
y = *(b'+1);


.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;


    ob = b; b = nb;


    free(ob); free(ob+1); }
```

RCU Reader/ Writer

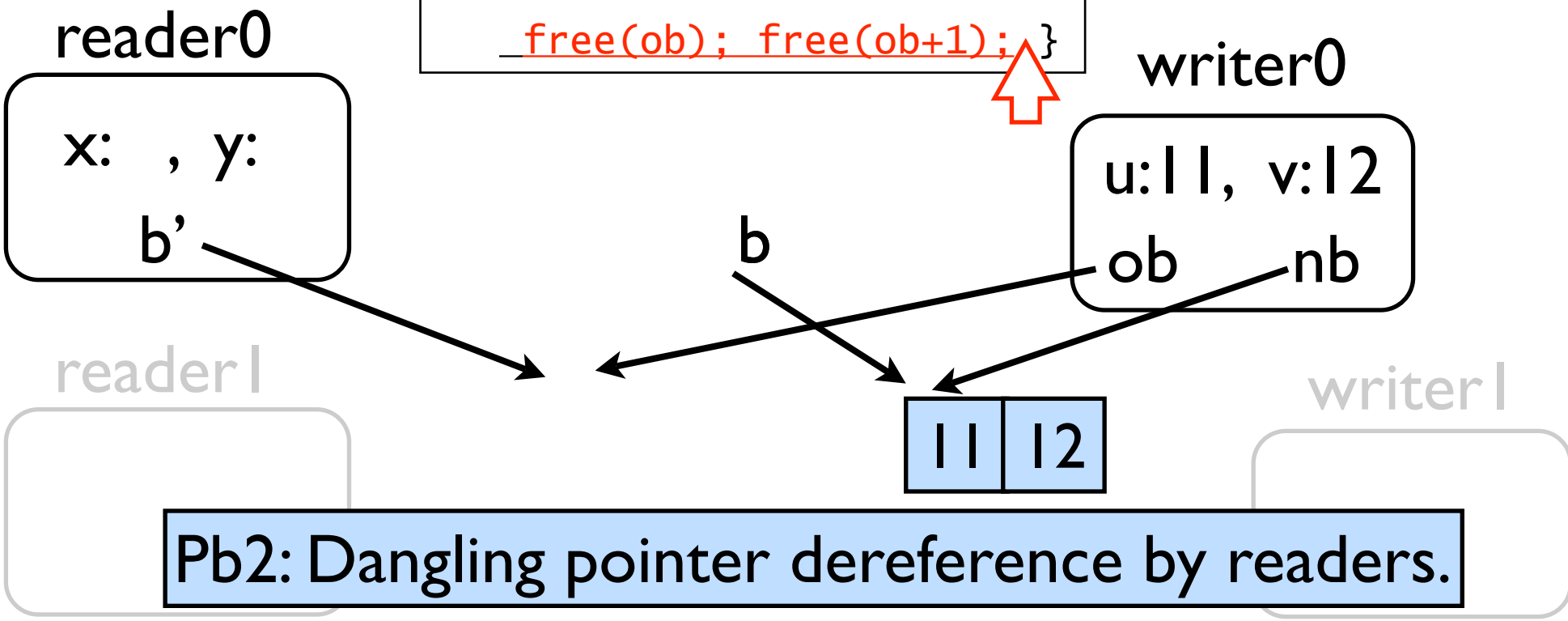


Pb2: Dangling pointer dereference by readers.

```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
   *nb = u; *(nb+1) = v;

   ob = b; b = nb;

 free(ob); free(ob+1); }
```

RCU Reader/ Writer



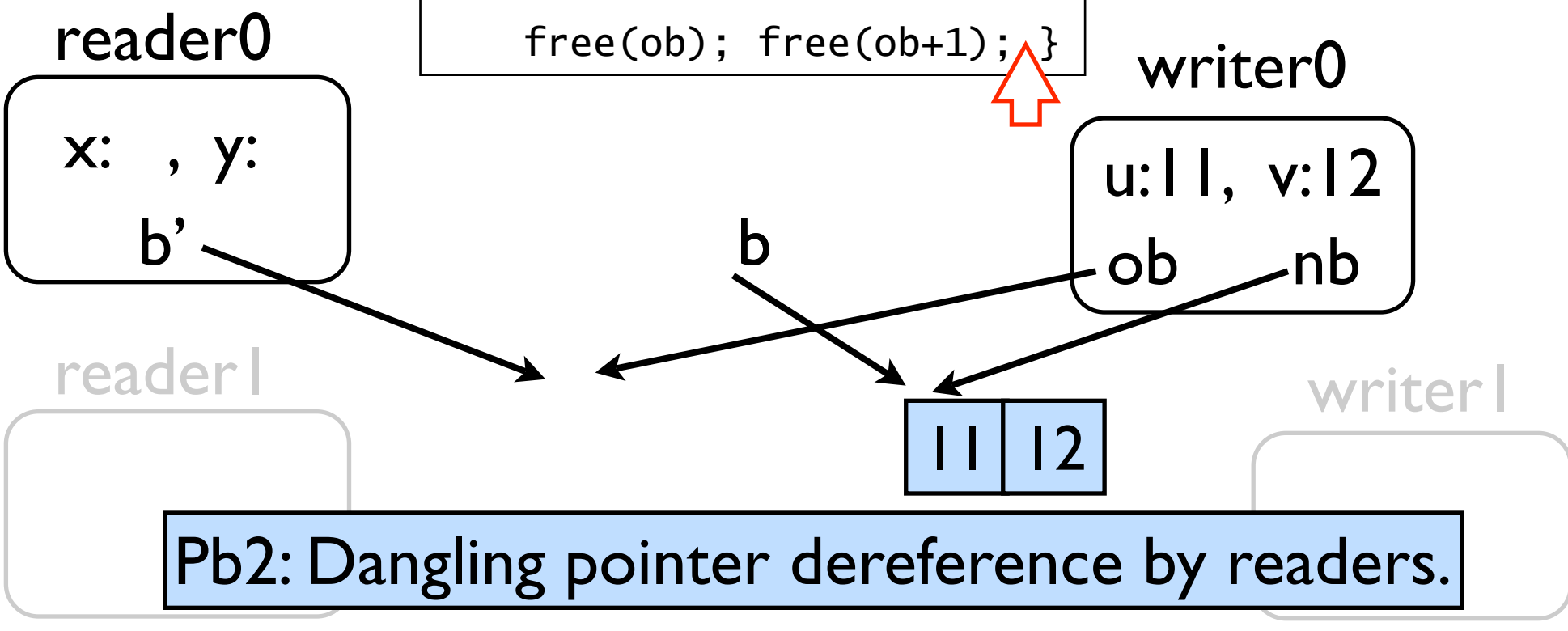
Pb2: Dangling pointer dereference by readers.

RCU Reader/ Writer

```
b' = b;
x = *b';
y = *(b'+1);

.. USE x,y ..
```

Dangling Pointer Dereference

```
.. PRODUCE u,v ..
nb = malloc(2);
                {
                b+1) = v;

ob = b; b = nb;

free(ob); free(ob+1); }
```



Pb2: Dangling pointer dereference by readers.

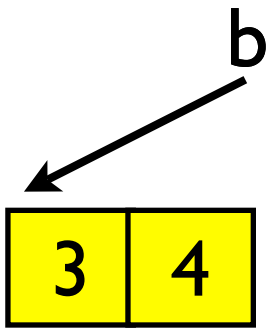
RCU Reader/ Writer

```
    b' = b;
    x = *b';
    y = *(b'+1);


.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;


    ob = b; b = nb;


   free(ob); free(ob+1); }
```

reader0

writer0



reader1

writer1

Pb2: Dangling pointer dereference by readers.

```
b' = b;
x = *b';
y = *(b'+1);


.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;


    ob = b; b = nb;
} sync {
    free(ob); free(ob+1); }
```

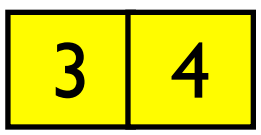
RCU Reader/

Wait until no readers are accessing the old buffer.

reader0

writer0

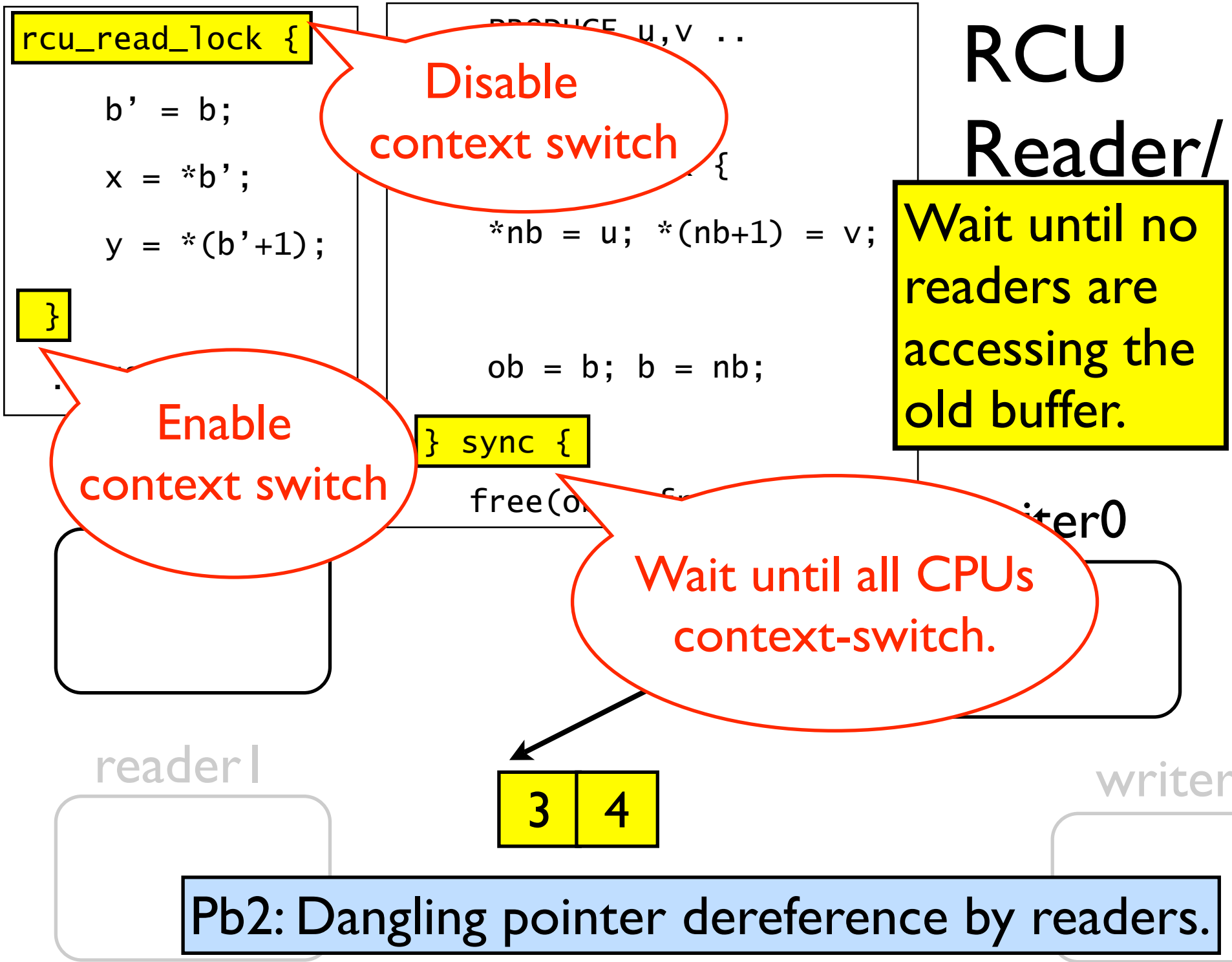
b



reader1

writer1

Pb2: Dangling pointer dereference by readers.


rcu_read_lock {
b' = b;
x = *b';
y = *(b'+1);
}
*nb = u; *(nb+1) = v;
ob = b; b = nb;
} sync {
free(o
Disable context switch
Enable context switch
Wait until all CPUs context-switch.
Wait until no readers are accessing the old buffer.
RCU Reader/
iter0
reader1
writer1
3
4
Pb2: Dangling pointer dereference by readers.

RCU Reader/

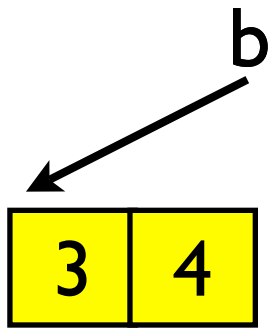
```
rcu_read_lock {
    b' = b;
    x = *b';
    y = *(b'+1);
}
 .. USE x,y ..
```

```
 .. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;


    ob = b; b = nb;
} sync {
    free(ob); free(ob+1); }
```

Wait until no readers are accessing the old buffer.

reader0

writer0



reader1

writer1

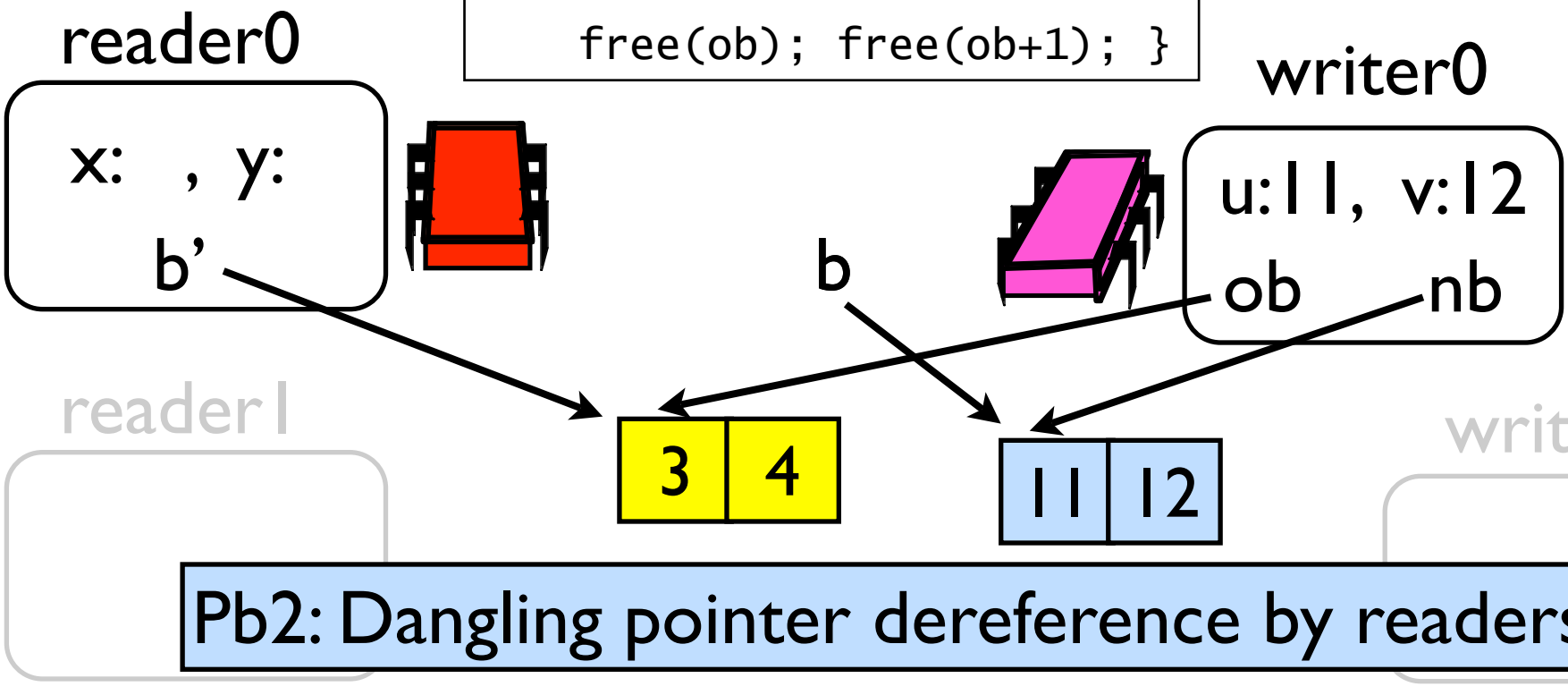
Pb2: Dangling pointer dereference by readers.

RCU Reader/

```
rcu_read_lock {
    b' = b;
    x = *b';
    y = *(b'+1);
}
  .. USE x,y ..
```

```
  .. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;


    ob = b; b = nb;
} sync {
    free(ob); free(ob+1); }
```

Wait until no readers are accessing the old buffer.



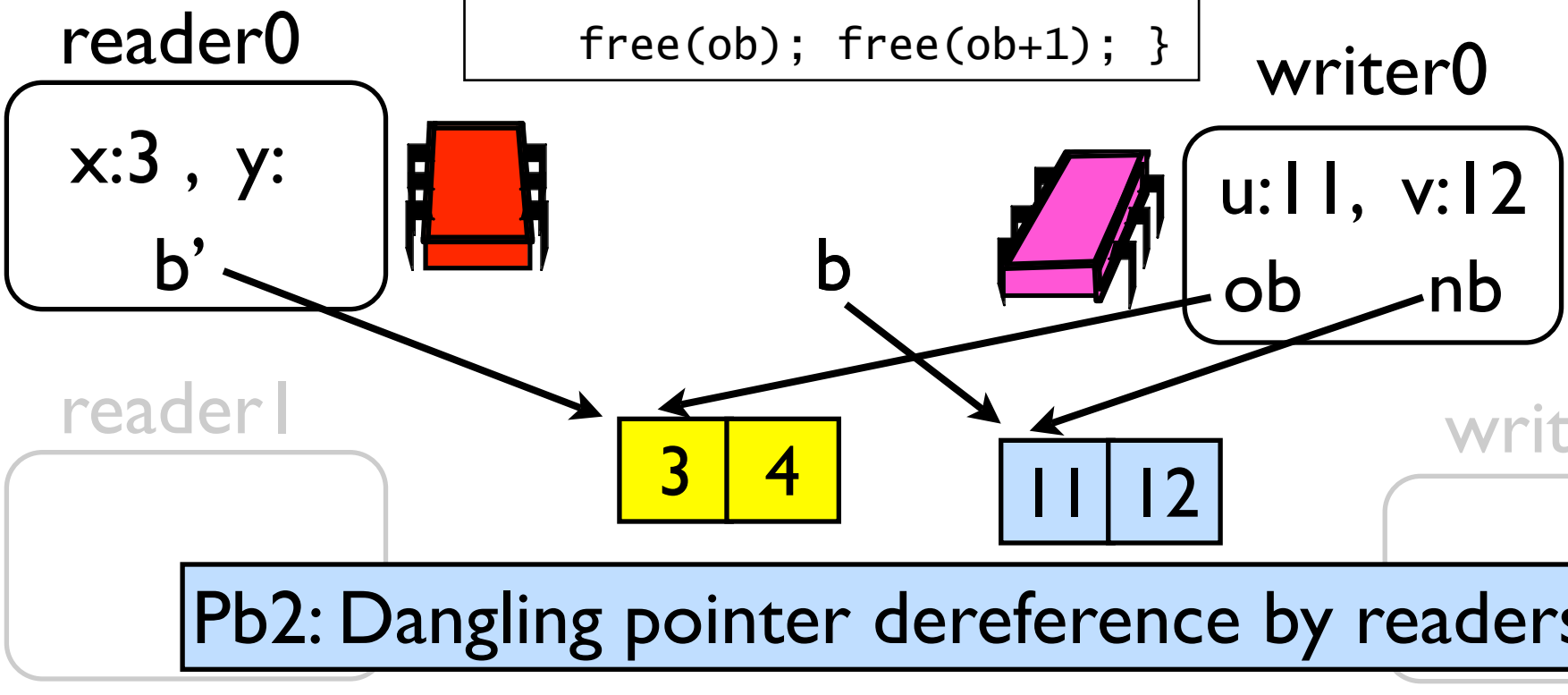
Pb2: Dangling pointer dereference by readers.

RCU Reader/

```
rcu_read_lock {
    b' = b;
    x = *b';
    y = *(b'+1);
}
  .. USE x,y ..
```

```
  .. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;

    ob = b; b = nb;
} sync {
    free(ob); free(ob+1); }
```

Wait until no readers are accessing the old buffer.



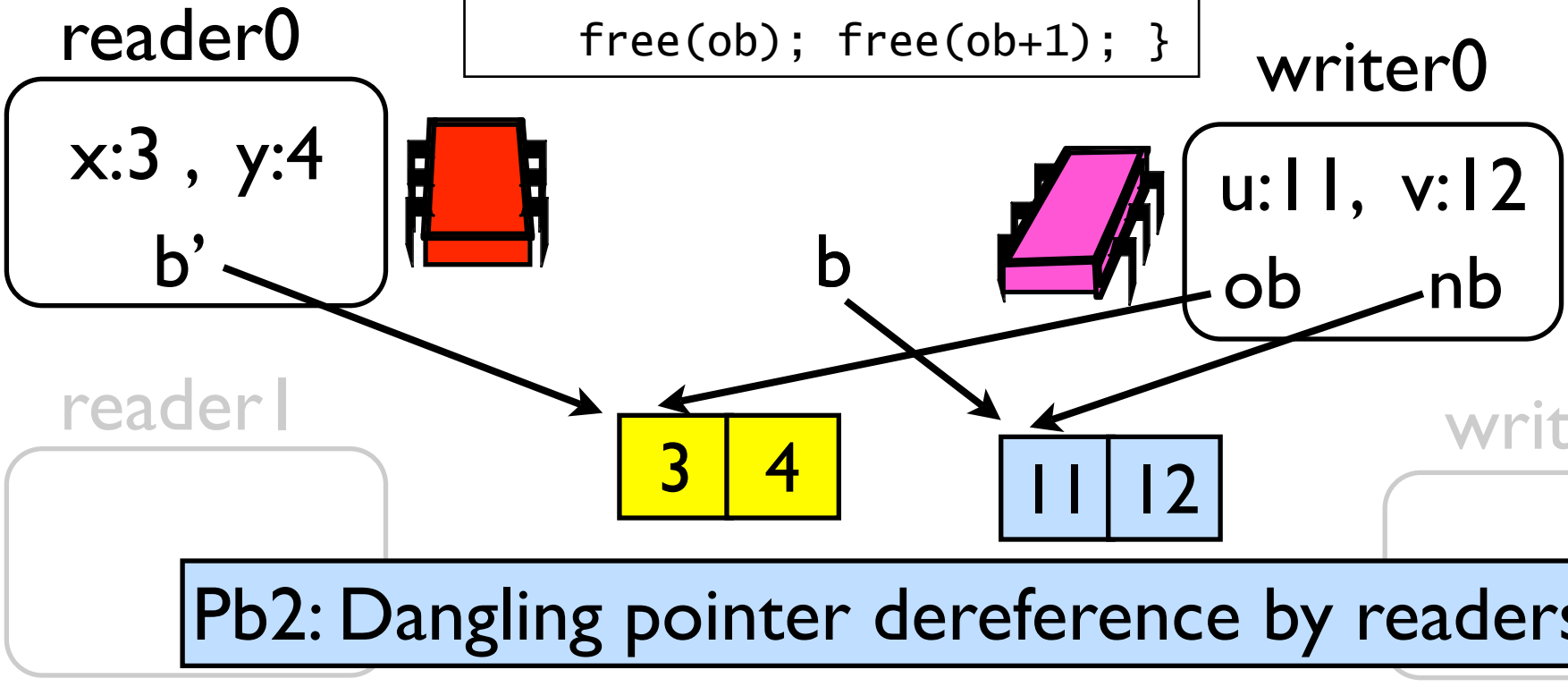
Pb2: Dangling pointer dereference by readers.

RCU Reader/

```
rcu_read_lock {
    b' = b;
    x = *b';
    y = *(b'+1);
}
  .. USE x,y ..
```

```
  .. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;
    ob = b; b = nb;
} sync {
    free(ob); free(ob+1); }
```

Wait until no readers are accessing the old buffer.



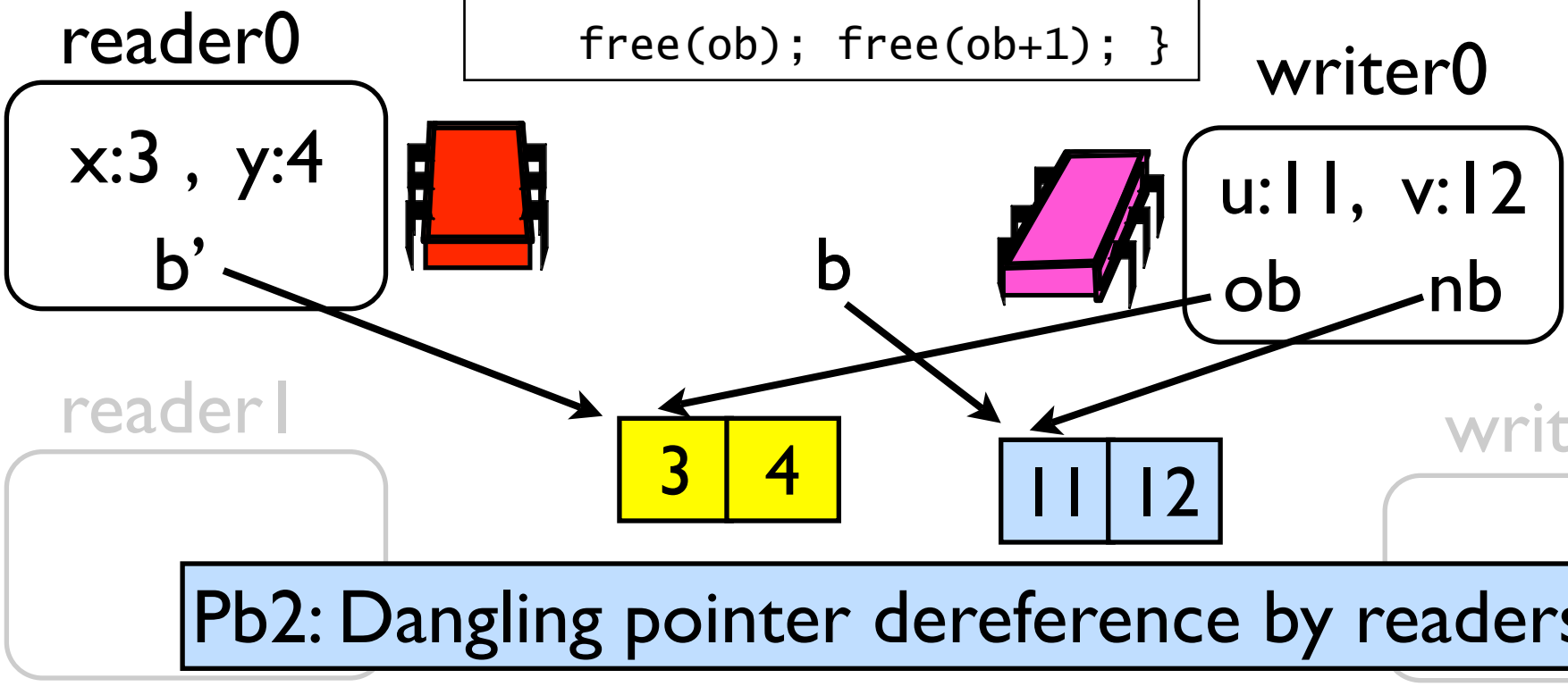
Pb2: Dangling pointer dereference by readers.

RCU Reader/

```
rcu_read_lock {
    b' = b;
    x = *b';
    y = *(b'+1);
}
.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;

    ob = b; b = nb;
} sync {
    free(ob); free(ob+1); }
```

Wait until no readers are accessing the old buffer.



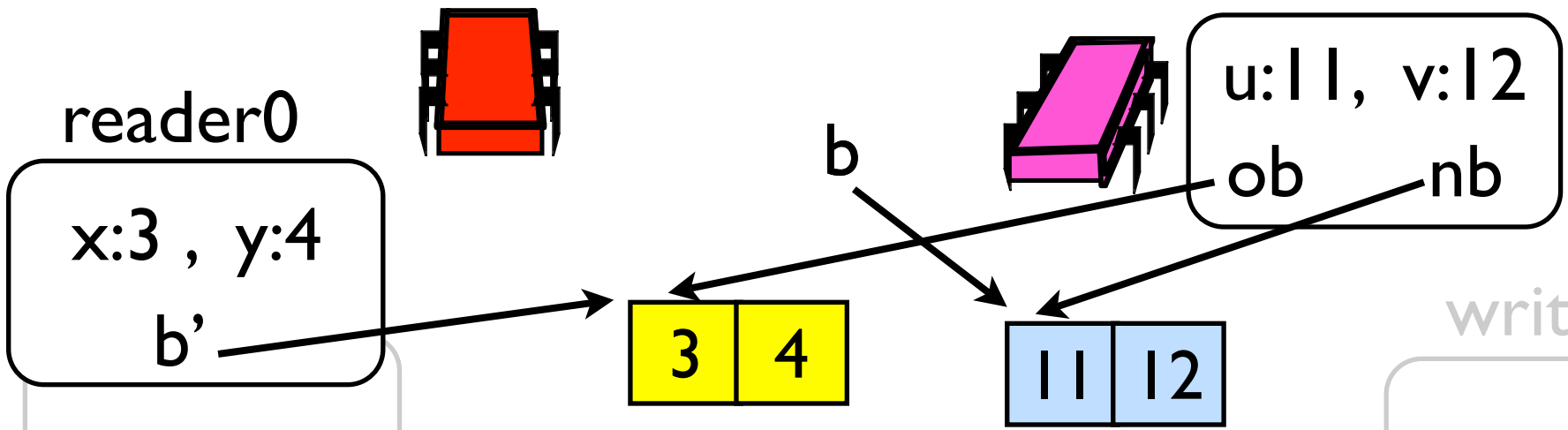
Pb2: Dangling pointer dereference by readers.

RCU Reader/

```
rcu_read_lock {
    b' = b;
    x = *b';
    y = *(b'+1);
}
.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;

    ob = b; b = nb;
} sync {
    free(ob); free(ob+1); }
```

Wait until no readers are accessing the old buffer.



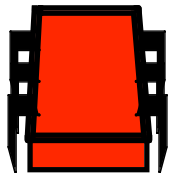
Pb2: Dangling pointer dereference by readers.

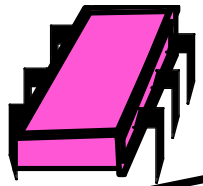
RCU Reader/

```
rcu_read_lock {
    b' = b;
    x = *b';
    y = *(b'+1);
}
.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;

    ob = b; b = nb;
} sync {
    free(ob); free(ob+1);}
```

Wait until no readers are accessing the old buffer.





writer0

u:11, v:12

ob nb

reader0

x:3 , y:4

b'

b

11 12

writer1

Pb2: Dangling pointer dereference by readers.

```
rcu_read_lock {
    b' = b;
    x = *b';
    y = *(b'+1);
}
.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;

    ob = b; b = nb;
} sync {
  free(ob); free(ob+1); }
```

RCU Reader/ Writer

```
rcu_read_lock {
    b' = b;
    x = *b';
    y = *(b'+1);
}
.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;
    barrier();
    ob = b; b = nb;
} sync {
  free(ob); free(ob+1); }
```

RCU Reader/ Writer

1) Handles weak memory.
2) Ensures that a reader never reads values from a non-initialized buffer.

```
rcu_read_lock {
    b' = b;
    x = *b';
    y = *(b'+1);
}
.. USE x,y ..
```

```
.. PRODUCE u,v ..
nb = malloc(2);
rcu_write_lock {
    *nb = u; *(nb+1) = v;
    barrier();
    ob = b; b = nb;
} sync {
   free(ob); free(ob+1); }
```

RCU Reader/ Writer

- RCU constructs:

  `rcu_read_lock { .. },`

  `rcu_write_lock { ... } sync { ... },  barrier`

- Linux API also includes asynchronous sync and

Research Issues

- Full specification of RCU programs.

Research Issues

- Full specification of RCU programs.
- Program logic for RCU.

Research Issues

- Full specification of RCU programs.
- Program logic for RCU.
- Automatic program analysis.
  - Shape analysis, termination analysis.

Research Issues

- Full specification of RCU programs.
- Program logic for RCU.
- Automatic program analysis.
  - Shape analysis, termination analysis.
- Verify the implementations of RCU.
  - Implementations of RCU, SRCU.
  - Prove correctness considering weak memory.

Research Issues

- Full specification of RCU programs.
- Program logic for RCU.
- Automatic program analysis.
  - Shape analysis, termination analysis.
- Verify the implementations of RCU.
  - Implementations of RCU, SRCU.
  - Prove correctness considering weak memory.
- New algorithms based on RCU.

Research Issues

- Full specification of RCU programs.
- Program logic for RCU.
- Automatic program analysis.
  - Shape analysis, termination analysis.
- Verify the implementations of RCU.
  - Implementations of RCU, SRCU.
  - Prove correctness considering weak memory.
- New algorithms based on RCU.

Overview of Logic

- Resource-invariant-based, thread-local reasoning.
- sync, barrier as ownership transfer operations.
- Reasoning about readers should use only those facts preserved by writers' operations.

Proof Sketch of RCU Readers/Writers

$\texttt{rcu\_read\_lock}\ \{$

$b' = b;$

$x = *b';$

$y = *(b' + 1);$

$\}$

$\texttt{rcu\_write\_lock}\ \{$

$*nb = u;\ *(nb + 1) = v;$

$\texttt{barrier}_{nb \mapsto \_,\_};$

$ob = b;\quad b = nb;$

$\}\ \texttt{sync}\ \{$

$\texttt{free}(ob); \texttt{free}(ob{+}1);$

$\}$

Proof Sketch of RCU Readers/Writers

$\texttt{rcu\_read\_lock}\ \{$

$b' = b;$

$x = *b';$

$y = *(b' + 1);$

$\}$

$\texttt{rcu\_write\_lock}\ \{$

$*nb = u;\ *(nb + 1) = v;$

$\texttt{barrier}_{nb \mapsto \_,\_};$

$ob = b;\quad b = nb;$

$\}\ \texttt{sync}\ \{$

$\texttt{free}(ob); \texttt{free}(ob{+}1);$

$\}$

Invariant-based, thread-local reasoning.

Proof Sketch of RCU Readers/Writers

$\{\text{emp}\}$
`rcu_read_lock` $\{$

$b' = b;$

$x = *b';$

$y = *(b' + 1);$

$\}$
$\{\text{emp}\}$

$\{nb \mapsto \_, \_\}$
`rcu_write_lock` $\{$

$*nb = u;\ *(nb + 1) = v;$

$\texttt{barrier}_{nb \mapsto \_,\_};$

$ob = b;\quad b = nb;$

$\}$ `sync` $\{$

$\texttt{free}(ob); \texttt{free}(ob{+}1);$

$\}$
$\{\text{emp}\}$

Invariant-based, thread-local reasoning.

Proof Sketch of RCU Readers/Writers

ResInv : $b \mapsto \_, \_$

$\{\text{emp}\}$
`rcu_read_lock {`

$\{nb \mapsto \_, \_\}$
`rcu_write_lock {`
$\{nb \mapsto \_, \_ * b \mapsto \_, \_\}$
$*nb = u;\ *(nb+1) = v;$

`barrier`$_{nb \mapsto \_,\_}$;

$ob = b$: $b = nb$:

ResInv holds initially for the shared heap.

`}`
$\{\text{emp}\}$

`} sync {`

`free`(ob);`free`$(ob{+}1)$;

`}`
$\{\text{emp}\}$

Invariant-based, thread-local reasoning.

Proof Sketch of RCU Readers/Writers

ResInv : $b \mapsto \_, \_$

$\{\text{emp}\}$
`rcu_read_lock {`

$b' = b;$

$x = *b';$

$y = *(b'+1)$

$\{$e

$\{nb \mapsto \_, \_\}$
`rcu_write_lock {`

$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$

$*nb = u;\ *(nb+1) = v;$

`barrier`$_{nb \mapsto \_,\_}$;

$ob = b;\quad b = nb;$

`sync {`

`free`(ob);`free`$(ob+1)$;

$\{\text{emp} * \boxed{b \mapsto \_, \_}\}$

}

$\{\text{emp}\}$

ResInv holds right before releasing the lock.

Invariant-based, thread-local reasoning.

Proof Sketch of RCU Readers/Writers

ResInv : $b \mapsto \_, \_$

$\{\mathsf{emp}\}$
`rcu_read_lock {`

$b' = b;$

$x = *b';$

$y = *(b' + 1);$

`}`
$\{\mathsf{emp}\}$

$\{nb \mapsto \_, \_\}$
`rcu_write_lock {`

$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
$*nb = u;\ *(nb + 1) = v;$
$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$

`barrier`$_{nb \mapsto \_,\_}$;

$ob = b;\quad b = nb;$

`} sync {`

`free`(ob);`free`$(ob{+}1)$;

$\{\mathsf{emp} * \boxed{b \mapsto \_, \_}\}$

`}`
$\{\mathsf{emp}\}$

Proof Sketch of RCU Readers/Writers

ResInv : $b \mapsto \_, \_$

$\{\text{emp}\}$
`rcu_read_lock {`

$b' = b;$

$\{nb \mapsto \_, \_\}$
`rcu_write_lock {`

$\{nb \mapsto \_, \_ * b \mapsto \_, \_\}$

$*nb = u; \ *(nb+1) = v;$

$\{nb \mapsto \_, \_ * b \mapsto \_, \_\}$

`barrier`$_{nb \mapsto \_,\_}$;

$\{\text{emp} * nb \mapsto \_, \_ * b \mapsto \_, \_\}$

$ob = b; \quad b = nb;$

barrier as ownership transfer from private to shared

`} sync {`

`free`(ob); `free`$(ob+1)$;

$\{\text{emp} * b \mapsto \_, \_\}$

`}`

$\{\text{emp}\}$

Proof Sketch of RCU Readers/Writers

ResInv : $b \mapsto \_, \_$

$\{\mathsf{emp}\}$
`rcu_read_lock {`

$b' = b;$

$x = *b';$

$y = *(b' + 1);$

`}`
$\{\mathsf{emp}\}$

$\{nb \mapsto \_, \_\}$
`rcu_write_lock {`
$\{nb \mapsto \_, \_ * b \mapsto \_, \_\}$
$*nb = u;\ *(nb+1) = v;$
$\{nb \mapsto \_, \_ * b \mapsto \_, \_\}$
`barrier`$_{nb \mapsto \_,\_}$`;`
$\{\mathsf{emp} * nb \mapsto \_, \_ * b \mapsto \_, \_\}$
$ob = b;\quad b = nb;$
$\{\mathsf{emp} * b \mapsto \_, \_ * ob \mapsto \_, \_\}$
`} sync {`

`free`(ob)`;free`$(ob+1)$`;`
$\{\mathsf{emp} * b \mapsto \_, \_\}$
`}`
$\{\mathsf{emp}\}$

Proof Sketch of RCU Readers/Writers

ResInv : $b \mapsto \_, \_$

$\{\text{emp}\}$
`rcu_read_lock {`

$b' = b;$

$x = *b';$

$y = *(b' + 1);$

`}`

$\{nb \mapsto \_, \_\}$
`rcu_write_lock {`
$\{nb \mapsto \_, \_ * b \mapsto \_, \_\}$
$*nb = u;\ *(nb + 1) = v;$
$\{nb \mapsto \_, \_ * b \mapsto \_, \_\}$
`barrier`$_{nb \mapsto \_,\_}$;
$\{\text{emp} * nb \mapsto \_, \_ * b \mapsto \_, \_\}$
$ob = b;\quad b = nb;$
$\{\text{emp} * b \mapsto \_, \_ * ob \mapsto \_, \_\}$
`} sync {`
$\{ob \mapsto \_, \_ * b \mapsto \_, \_\}$
`free`(ob);`free`$(ob+1)$;
$\{\text{emp} * b \mapsto \_, \_\}$
`}`
$\{\text{emp}\}$

sync as ownership transfer from shared to private

Proof Sketch of RCU Readers/Writers

ResInv : $b \mapsto \_, \_$

$\{\mathsf{emp}\}$
`rcu_read_lock {`

$b' = b;$

$x = *b';$

$y = *(b' + 1);$

`}`
$\{\mathsf{emp}\}$

$\{nb \mapsto \_, \_\}$
`rcu_write_lock {`
$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
$*nb = u;\ *(nb+1) = v;$
$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
`barrier`$_{nb \mapsto \_,\_}$;
$\{\mathsf{emp} * \boxed{nb \mapsto \_, \_ * b \mapsto \_, \_}\}$
$ob = b;\quad b = nb;$
$\{\mathsf{emp} * \boxed{b \mapsto \_, \_ * ob \mapsto \_, \_}\}$
`} sync {`
$\{ob \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
`free`(ob);`free`$(ob{+}1)$;
$\{\mathsf{emp} * \boxed{b \mapsto \_, \_}\}$
`}`
$\{\mathsf{emp}\}$

Proof Sketch of RCU Readers/Writers

ResInv : $b \mapsto \_, \_$

$\{\text{emp}\}$
`rcu_read_lock {`

$b' = b;$

$x = *b';$

$y = *(b' + 1);$

`}`
$\{\text{emp}\}$

$\{nb \mapsto \_, \_\}$
`rcu_write_lock {`
$\{nb \mapsto \_, \_ * b \mapsto \_, \_\}$
$*nb = u;\ *(nb + 1) = v;$
$\{nb \mapsto \_, \_ * b \mapsto \_, \_\}$
`barrier`$_{nb \mapsto \_,\_}$;
$\{\text{emp} * nb \mapsto \_, \_ * b \mapsto \_, \_\}$
$ob = b;\quad b = nb;$
$\{\text{emp} * b \mapsto \_, \_ * ob \mapsto \_, \_\}$
`} sync {`
$\{ob \mapsto \_, \_ * b \mapsto \_, \_\}$
`free`(ob);`free`$(ob+1)$;
$\{\text{emp} * b \mapsto \_, \_\}$
`}`
$\{\text{emp}\}$

AlwaysResInv : $b \mapsto \_, \_ * \text{true}$

Proof Sketch of RCU Readers/Writers

$\{\text{emp}\}$
`rcu_read_lock {`

$b' = b;$

$x = *b';$

$y = *(b' + 1);$

`}`
$\{\text{emp}\}$

$\{nb \mapsto \_, \_\}$
`rcu_write_lock {`
$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
$*nb = u;\ *(nb+1) = v;$
$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
`barrier`$_{nb \mapsto \_,\_}$;
$\{\text{emp} * \boxed{nb \mapsto \_, \_ * b \mapsto \_, \_}\}$
$ob = b;\quad b = nb;$
$\{\text{emp} * \boxed{b \mapsto \_, \_ * ob \mapsto \_, \_}\}$
`} sync {`
$\{ob \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
`free`(ob);`free`$(ob{+}1)$;
$\{\text{emp} * \boxed{b \mapsto \_, \_}\}$
`}`
$\{\text{emp}\}$

ResInv : $b \mapsto \_, \_$

AlwaysResInv : $b \mapsto \_, \_ * \text{true}$

Pres : $\forall a.\ (\boxed{a \mapsto \_, \_ * \text{true}} \rightsquigarrow \boxed{a \mapsto \_, \_ * \text{true}})$

Proof Sketch of RCU Readers/Writers

ResInv : $b \mapsto \_, \_$

$\{\text{emp}\}$
`rcu_read_lock {`
$\{\text{emp} * \boxed{b \mapsto \_, \_ * \text{true}}\}$
$b' = b;$
$x = *b';$
$y = *(b' + 1);$
`}`
$\{\text{emp}\}$

Start with AlwaysResInv.

$\{nb \mapsto \_, \_\}$
`rcu_write_lock {`
$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
$*nb = u;\ *(nb + 1) = v;$
$* \boxed{b \mapsto \_, \_}\}$
$\{nb \mapsto \_, \_;$
$\boxed{nb \mapsto \_, \_ * b \mapsto \_, \_}\}$
$ob = b;\quad b = nb;$
$\{\text{emp} * \boxed{b \mapsto \_, \_ * ob \mapsto \_, \_}\}$
`} sync {`
$\{ob \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
$\texttt{free}(ob); \texttt{free}(ob+1);$
$\{\text{emp} * \boxed{b \mapsto \_, \_}\}$
`}`
$\{\text{emp}\}$

AlwaysResInv : $b \mapsto \_, \_ * \text{true}$

Pres : $\forall a. \left(\boxed{a \mapsto \_, \_ * \text{true}} \rightsquigarrow \boxed{a \mapsto \_, \_ * \text{true}}\right)$

Proof Sketch of RCU Readers/Writers

ResInv : $b \mapsto \_, \_$

$\{\mathsf{emp}\}$
`rcu_read_lock {`
$\{\mathsf{emp} * b \mapsto \_, \_ * \mathsf{true}\}$
$b' = b;$
$x = *b';$
$y = *(b' + 1);$
$\{\mathsf{emp} * b' \mapsto \_, \_ * \mathsf{true}\}$
`}`
$\{\mathsf{emp}\}$

$\{nb \mapsto \_, \_\}$
`rcu_write_lock {`
$\{nb \mapsto \_, \_ * b \mapsto \_, \_\}$
$*nb = u;\ *(nb + 1) = v;$
$\{nb \mapsto \_, \_ * b \mapsto \_, \_\}$
`barrier`$_{nb \mapsto \_,\_}$;
$\{\mathsf{emp} * nb \mapsto \_, \_ * b \mapsto \_, \_\}$
$ob = b;\quad b = nb;$
$\{\mathsf{emp} * b \mapsto \_, \_ * ob \mapsto \_, \_\}$
`} sync {`
$\_, \_ * b \mapsto \_, \_\}$
$ob);$ `free`$(ob+1);$
$b \mapsto \_, \_\}$

Can end with any assertion.

AlwaysResInv : $b \mapsto \_, \_ * \mathsf{true}$

Pres : $\forall a.\ (a \mapsto \_, \_ * \mathsf{true} \rightsquigarrow a \mapsto \_, \_ * \mathsf{true})$

Proof Sketch of RCU Readers/Writers

$\{\mathsf{emp}\}$
`rcu_read_lock {`
$\{\mathsf{emp} * \boxed{b \mapsto \_, \_} * \mathsf{true}\}$
$b' = b;$
$\{\mathsf{emp} * \boxed{b' \mapsto \_, \_} * \mathsf{true}\}$
$x = *b';$

$y = *(b' + 1);$
$\{\mathsf{emp} * \boxed{b' \mapsto \_, \_} * \mathsf{true}\}$
`}`
$\{\mathsf{emp}\}$

$\{nb \mapsto \_, \_\}$
`rcu_write_lock {`
$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
$*nb = u;\ *(nb+1) = v;$
$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
$\texttt{barrier}_{nb \mapsto \_, \_};$
$\{\mathsf{emp} * \boxed{nb \mapsto \_, \_ * b \mapsto \_, \_}\}$
$ob = b;\quad b = nb;$
$\{\mathsf{emp} * \boxed{b \mapsto \_, \_ * ob \mapsto \_, \_}\}$
`} sync {`
$\{ob \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
$\texttt{free}(ob); \texttt{free}(ob+1);$
$\{\mathsf{emp} * \boxed{b \mapsto \_, \_}\}$
`}`
$\{\mathsf{emp}\}$

ResInv : $\boxed{b \mapsto \_, \_}$

AlwaysResInv : $\boxed{b \mapsto \_, \_} * \mathsf{true}$

Pres : $\forall a.\ (\boxed{a \mapsto \_, \_} * \mathsf{true} \rightsquigarrow \boxed{a \mapsto \_, \_} * \mathsf{true})$

Proof Sketch of RCU [illegible] Writers

Not modify the shared heap.

$b \mapsto \_, \_$

$\{\text{emp}\}$
`rcu_read_lock {`
$\{\text{emp} * \boxed{b \mapsto \_, \_ * \text{true}}\}$
$b' = b;$
$\{\text{emp} * \boxed{b' \mapsto \_, \_ * \text{true}}\}$
$x = *b';$

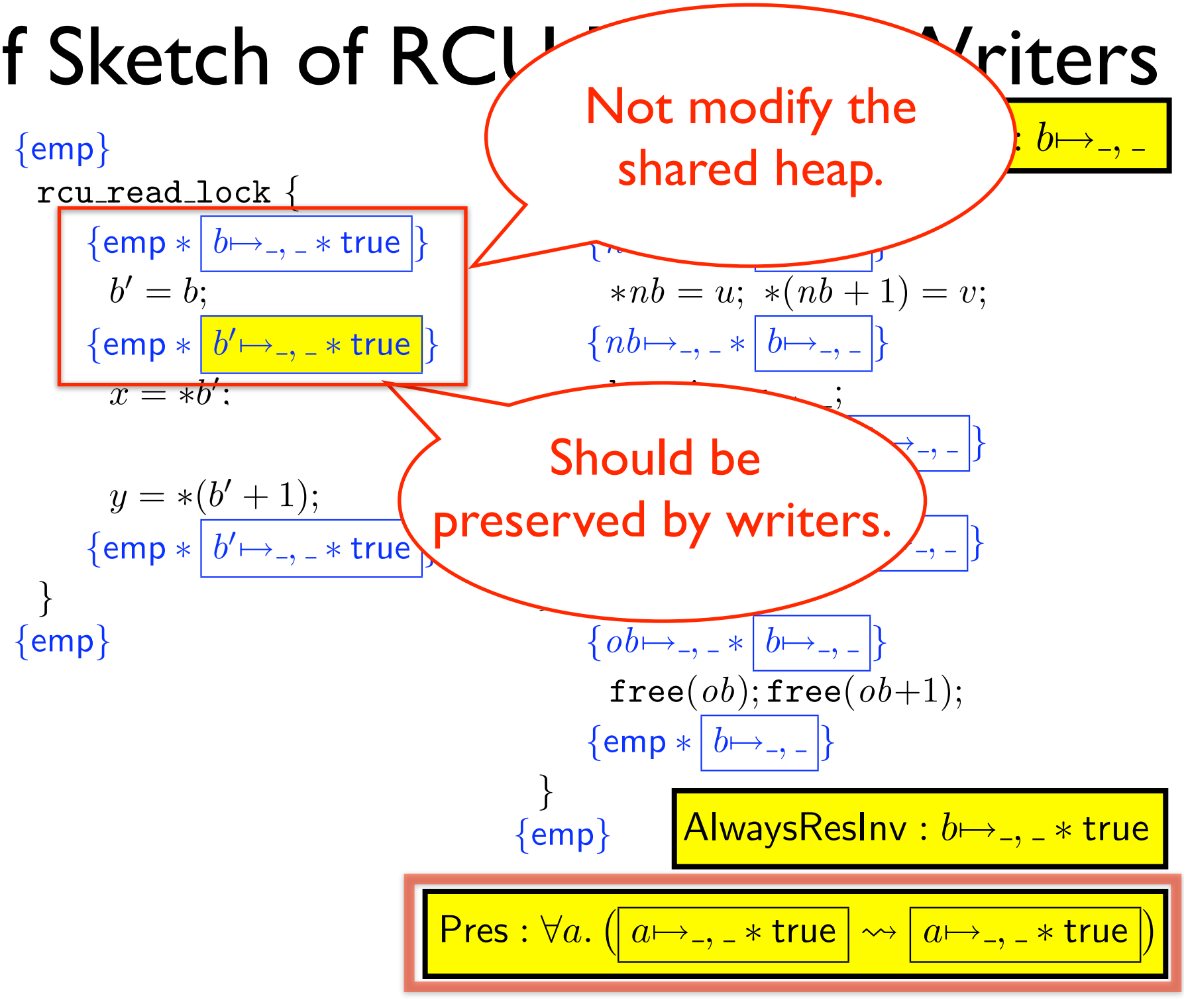
$y = *(b' + 1);$
$\{\text{emp} * \boxed{b' \mapsto \_, \_ * \text{true}}\}$
`}`
$\{\text{emp}\}$

$*nb = u;\ *(nb+1) = v;$
$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
$\texttt{barrier}_{nb \mapsto \_, \_};$
$\{\text{emp} * \boxed{nb \mapsto \_, \_ * b \mapsto \_, \_}\}$
$ob = b;\quad b = nb;$
$\{\text{emp} * \boxed{b \mapsto \_, \_ * ob \mapsto \_, \_}\}$
`} sync {`
$\{ob \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
$\texttt{free}(ob); \texttt{free}(ob+1);$
$\{\text{emp} * \boxed{b \mapsto \_, \_}\}$
`}`
$\{\text{emp}\}$

AlwaysResInv : $\boxed{b \mapsto \_, \_ * \text{true}}$

Pres : $\forall a.\ (\boxed{a \mapsto \_, \_ * \text{true}} \rightsquigarrow \boxed{a \mapsto \_, \_ * \text{true}})$

Proof Sketch of RCU [illegible] Writers



Proof Sketch of RCU Readers/Writers

$\{\mathsf{emp}\}$
`rcu_read_lock {`
$\{\mathsf{emp} * \boxed{b \mapsto \_, \_ * \mathsf{true}}\}$
$b' = b;$
$\{\mathsf{emp} * \boxed{b' \mapsto \_, \_ * \mathsf{true}}\}$
$x = *b';$
$\{\mathsf{emp} * \boxed{b' \mapsto \_, \_ * \mathsf{true}}\}$
$y = *(b' + 1);$
$\{\mathsf{emp} * \boxed{b' \mapsto \_, \_ * \mathsf{true}}\}$
`}`
$\{\mathsf{emp}\}$

$\{nb \mapsto \_, \_\}$
`rcu_write_lock {`
$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
$*nb = u;\ *(nb+1) = v;$
$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
$\texttt{barrier}_{nb \mapsto \_, \_};$
$\{\mathsf{emp} * \boxed{nb \mapsto \_, \_ * b \mapsto \_, \_}\}$
$ob = b;\quad b = nb;$
$\{\mathsf{emp} * \boxed{b \mapsto \_, \_ * ob \mapsto \_, \_}\}$
`} sync {`
$\{ob \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
$\texttt{free}(ob); \texttt{free}(ob+1);$
$\{\mathsf{emp} * \boxed{b \mapsto \_, \_}\}$
`}`
$\{\mathsf{emp}\}$

ResInv : $\boxed{b \mapsto \_, \_}$

AlwaysResInv : $\boxed{b \mapsto \_, \_ * \mathsf{true}}$

Pres : $\forall a.\ \left(\boxed{a \mapsto \_, \_ * \mathsf{true}} \rightsquigarrow \boxed{a \mapsto \_, \_ * \mathsf{true}}\right)$

Proof Sketch of RCU Readers/Writers

$\{\mathsf{emp}\}$
`rcu_read_lock {`
$\{\mathsf{emp} * \boxed{b \mapsto \_, \_ * \mathsf{true}}\}$
$b' = b;$
$\{\mathsf{emp} * \boxed{b' \mapsto \_, \_ * \mathsf{true}}\}$
$x = *b';$
$\{\mathsf{emp} * \boxed{b' \mapsto \_, \_ * \mathsf{true}}\}$
$y = *(b' + 1);$
$\{\mathsf{emp} * \boxed{b' \mapsto \_, \_ * \mathsf{true}}\}$
`}`
$\{\mathsf{emp}\}$

$\{nb \mapsto \_, \_\}$
`rcu_write_lock {`
$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
$*nb = u;\ *(nb + 1) = v;$
$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
`barrier`$_{nb \mapsto \_,\_}$;
$\{\mathsf{emp} * \boxed{nb \mapsto \_, \_ * b \mapsto \_, \_}\}$
$ob = b;\quad b = nb;$
$\{\mathsf{emp} * \boxed{b \mapsto \_, \_ * ob \mapsto \_, \_}\}$
`} sync {`
$\{ob \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
`free`(ob);`free`$(ob+1)$;
$\{\mathsf{emp} * \boxed{b \mapsto \_, \_}\}$
`}`
$\{\mathsf{emp}\}$

ResInv : $\boxed{b \mapsto \_, \_}$

AlwaysResInv : $\boxed{b \mapsto \_, \_ * \mathsf{true}}$

Pres : $\forall a.\ \left(\boxed{a \mapsto \_, \_ * \mathsf{true}} \rightsquigarrow \boxed{a \mapsto \_, \_ * \mathsf{true}}\right)$

Proof Sketch of RCU Readers/Writers

ResInv : $b \mapsto \_, \_$

$\{\mathsf{emp}\}$
`rcu_read_lock {`
$\{\mathsf{emp} * \boxed{b \mapsto \_, \_ * \mathsf{true}}\}$
$b' = b;$
$\{\mathsf{emp} * \boxed{b' \mapsto \_, \_ * \mathsf{true}}\}$
$x = *b';$
$\{\mathsf{emp} * \boxed{b' \mapsto \_, \_ * \mathsf{true}}\}$
$y = *(b' + 1);$
$\{\mathsf{emp} * \boxed{b' \mapsto \_, \_ * \mathsf{true}}\}$
`}`
$\{\mathsf{emp}\}$

$\{nb \mapsto \_, \_\}$
`rcu_write_lock {`
$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
$*nb = u;\ *(nb + 1) = v;$
$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
`barrier`$_{nb \mapsto \_, \_};$
$\{\mathsf{emp} * \boxed{nb \mapsto \_, \_ * b \mapsto \_, \_}\}$
$ob = b;\quad b = nb;$
$\{\mathsf{emp} * \boxed{b \mapsto \_, \_ * ob \mapsto \_, \_}\}$
`} sync {`
$\{ob \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
`free`$(ob);$`free`$(ob{+}1);$
$\{\mathsf{emp} * \boxed{b \mapsto \_, \_}\}$
`}`
$\{\mathsf{emp}\}$

AlwaysResInv : $b \mapsto \_, \_ * \mathsf{true}$

Pres : $\forall a.\ (\boxed{a \mapsto \_, \_ * \mathsf{true}} \rightsquigarrow \boxed{a \mapsto \_, \_ * \mathsf{true}})$

Proof Sketch of RCU Readers/Writers

ResInv : $b \mapsto \_, \_$

$\{\text{emp}\}$
rcu_read_lock {
$\{\text{emp} * \boxed{b \mapsto \_, \_ * \text{true}}\}$
$b' = b;$
$\{\text{emp} * \boxed{b' \mapsto \_, \_ * \text{true}}\}$

$\{nb \mapsto \_, \_\}$
rcu_write_lock {
$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
$*nb = u;\ *(nb+1) = v;$
$\{nb \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$

1) Invariant-based, thread-local reasoning.
2) barrier, sync as ownership transfer operations.
3) Use only preserved facts, when verifying readers.

}
$\{\text{emp}\}$

sync {
$\{ob \mapsto \_, \_ * \boxed{b \mapsto \_, \_}\}$
free(ob); free($ob+1$);
$\{\text{emp} * \boxed{b \mapsto \_, \_}\}$
}
$\{\text{emp}\}$

AlwaysResInv : $b \mapsto \_, \_ * \text{true}$

Pres : $\forall a.\ (\boxed{a \mapsto \_, \_ * \text{true}} \rightsquigarrow \boxed{a \mapsto \_, \_ * \text{true}})$

Assertion Language

$$P ::= \text{true} \mid E \mapsto F \mid \text{emp} \mid P * Q \mid \ldots\ldots$$

$$K ::= P \mid \boxed{P} \mid K * L \mid \ldots\ldots$$

- P describes the heap local to the thread.
- $\boxed{P}$ describes the shared heap.
- K describes both.

Judgments

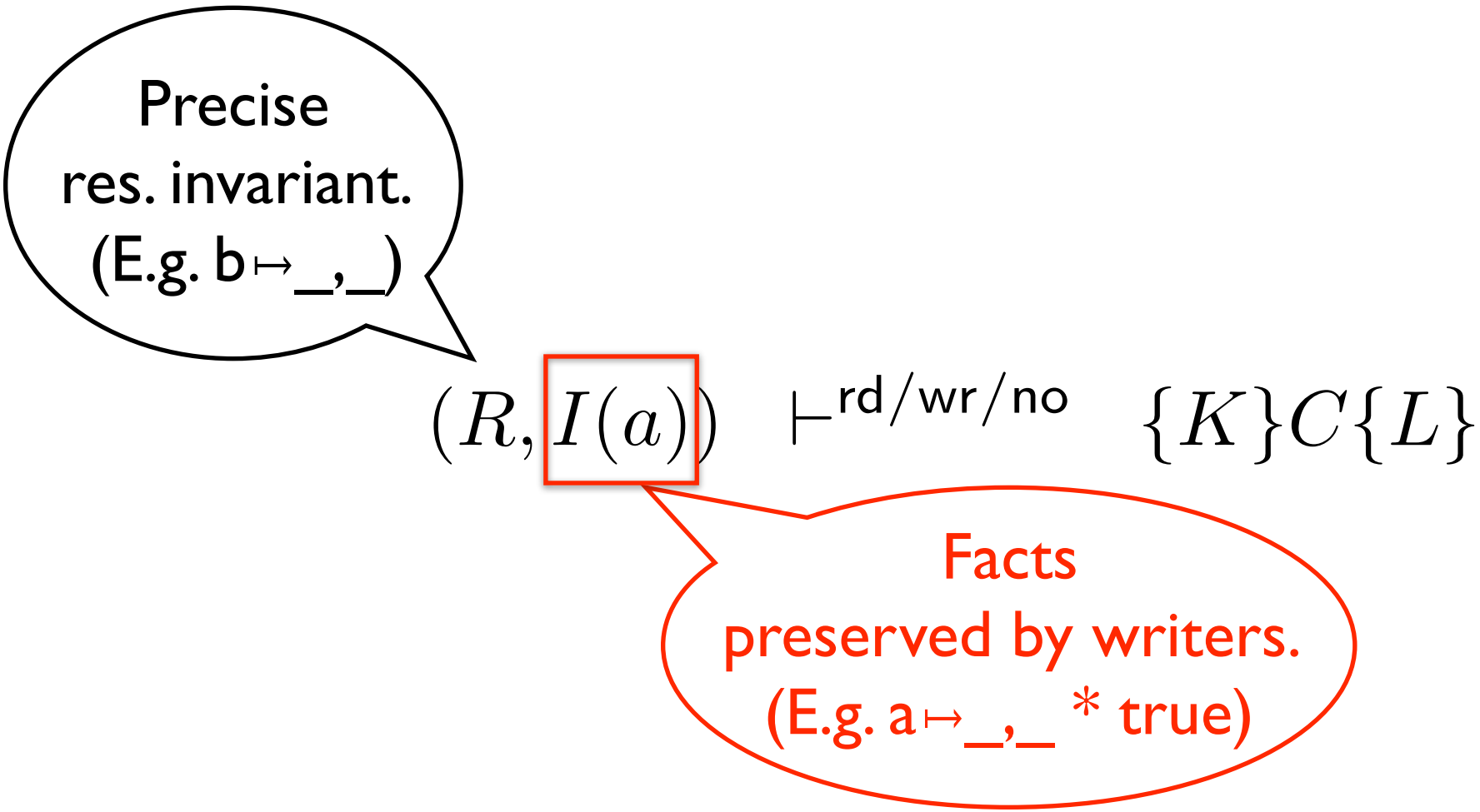
$$(R, I(a)) \quad \vdash^{\text{rd/wr/no}} \quad \{K\}C\{L\}$$

Judgments

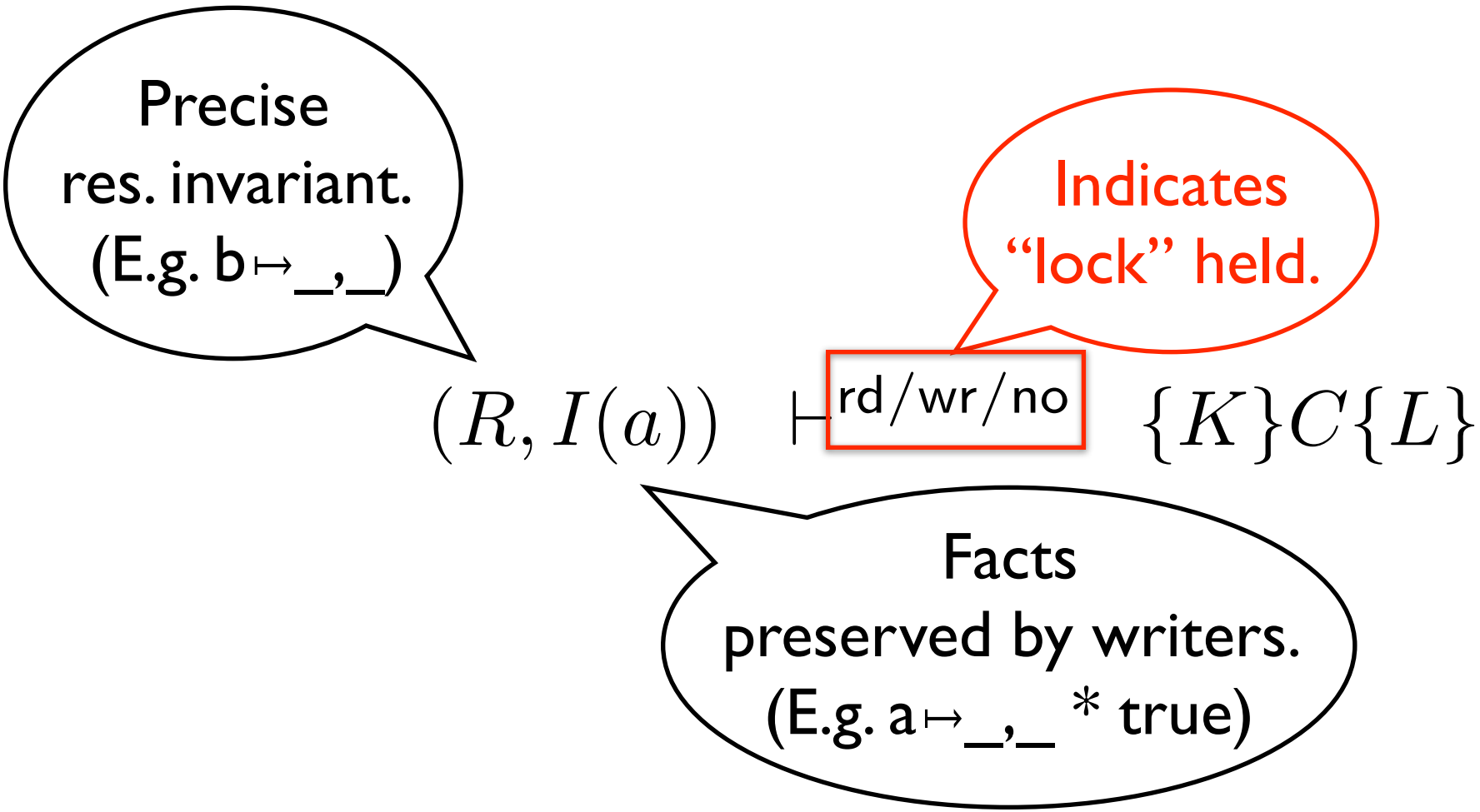
Precise
res. invariant.
(E.g. b↦_,_)

$$(R, I(a)) \quad \vdash^{\text{rd/wr/no}} \quad \{K\}C\{L\}$$

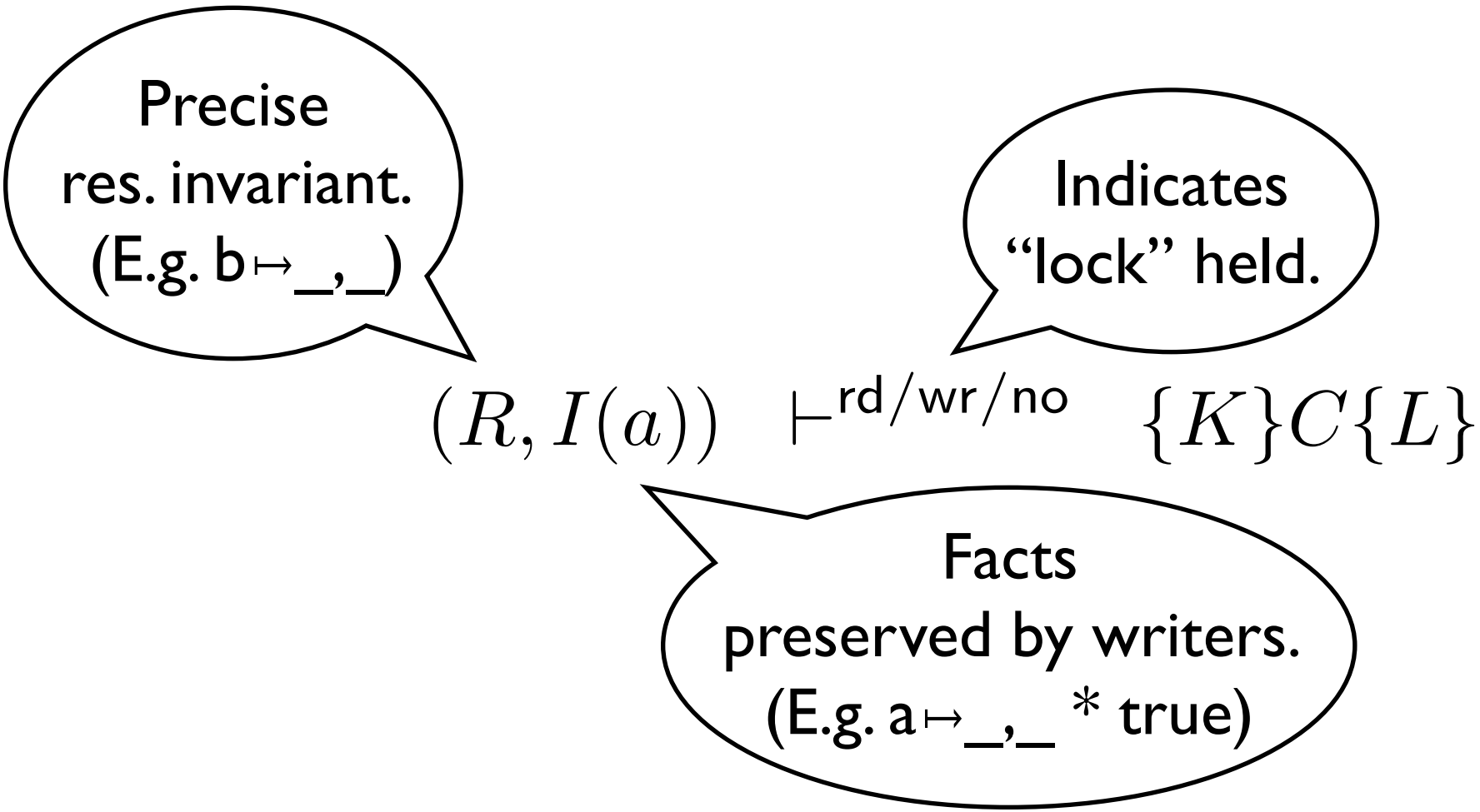
Judgments


Precise
res. invariant.
(E.g. b↦_,_)
$(R, I(a))$
$\vdash^{\text{rd/wr/no}}$
$\{K\}C\{L\}$
Facts
preserved by writers.
(E.g. a↦_,_ * true)

Judgments


Precise
res. invariant.
(E.g. b↦_,_)
Indicates
“lock” held.
$(R, I(a))$
$\vdash$
rd/wr/no
$\{K\}C\{L\}$
Facts
preserved by writers.
(E.g. a↦_,_ * true)

Judgments


Precise
res. invariant.
(E.g. b↦_,_)
Indicates
“lock” held.
$(R, I(a))$ $\vdash^{\text{rd/wr/no}}$ $\{K\}C\{L\}$
Facts
preserved by writers.
(E.g. a↦_,_ * true)

Proof Rule for Lock/Sync

$$\frac{(R, I(a)) \vdash^{\mathsf{wr}} \{P * \boxed{R}\} C \{Q' * \boxed{R' * R}\} \qquad (R, I(a)) \vdash^{\mathsf{wr}} \{Q' * R' * \boxed{R}\} D \{Q * \boxed{R}\}}{(R, I(a)) \vdash^{\mathsf{no}} \{P\} \texttt{rcu\_write\_lock}\ C\ \texttt{sync}\ D \{Q\}}$$

Proof Rule for Lock/Sync

Allowed to access the shared heap described by R.

$$\frac{(R, I(a)) \vdash^{\mathsf{wr}} \{P * \boxed{R}\} C \{Q' * \boxed{R' * R}\} \quad (R, I(a)) \vdash^{\mathsf{wr}} \{Q' * R' * \boxed{R}\} D \{Q * \boxed{R}\}}{(R, I(a)) \vdash^{\mathsf{no}} \{P\} \texttt{rcu\_write\_lock}\ C\ \texttt{sync}\ D \{Q\}}$$

Proof Rule for Lock/Sync

$$\frac{(R, I(a)) \vdash^{\mathsf{wr}} \{P * \boxed{R}\} C \{Q' * \boxed{R' * R}\} \qquad (R, I(a)) \vdash^{\mathsf{wr}} \{Q' * R' * \boxed{R}\} D \{Q * \boxed{R}\}}{(R, I(a)) \vdash^{\mathsf{no}} \{P\}\texttt{rcu\_write\_lock}\ C\ \boxed{\texttt{sync}}\ D\{Q\}}$$

sync transfers R' from shared to private.

Proof Rule for Lock/Sync

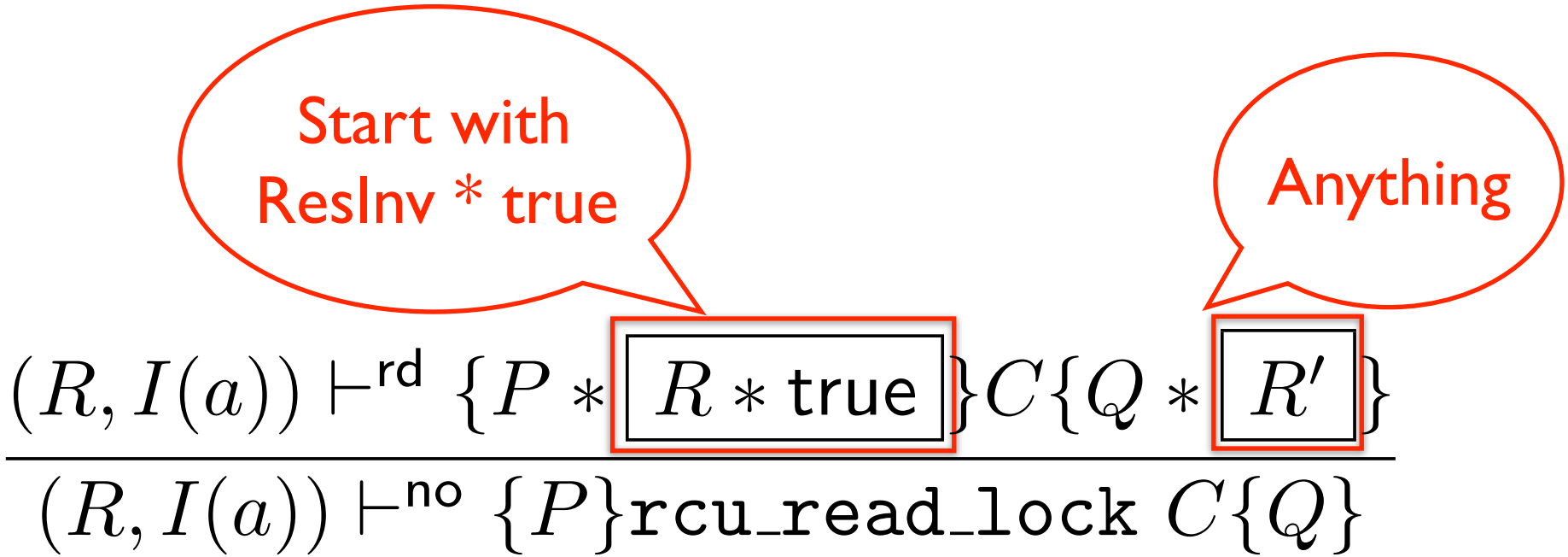
Every op in C and D should preserve R*true and I(a).

$$\frac{(R, I(a)) \vdash^{\mathsf{wr}} \{P * \boxed{R}\} C \{Q' * \boxed{R' * R}\} \quad (R, I(a)) \vdash^{\mathsf{wr}} \{Q' * R' * \boxed{R}\} D \{Q * \boxed{R}\}}{(R, I(a)) \vdash^{\mathsf{no}} \{P\}\texttt{rcu\_write\_lock}\ C\ \texttt{sync}\ D\{Q\}}$$

Proof Rule for Readers

$$\frac{(R, I(a)) \vdash^{\mathsf{rd}} \{P * \boxed{R * \mathsf{true}}\} C \{Q * \boxed{R'}\}}{(R, I(a)) \vdash^{\mathsf{no}} \{P\} \texttt{rcu\_read\_lock}\ C \{Q\}}$$

Proof Rule for Readers



Proof Rule for Readers

Every op in C can only read the shared.

$$\frac{(R, I(a)) \vdash^{\mathsf{rd}} \{P * \boxed{R * \mathsf{true}}\} C \{Q * \boxed{R'}\}}{(R, I(a)) \vdash^{\mathsf{no}} \{P\}\texttt{rcu\_read\_lock}\ C\{Q\}}$$

Proof Rule

$$(R, I(a)) \vdash^{\mathsf{wr}} \{P * Q * \boxed{R}\}\texttt{barrier}_{Q}\{P * \boxed{Q * R}\}$$

Transfer Q from private to shared.

Q has to be precise.

Soundness

- No weak memory yet.
- Proved by the compilation into RGSep by Vafeiadis and Parkinson.

Messages

- Sync transfers a heaplet from shared to private.
- Barrier transfers a heaplet from private to shared.

RCU Reader/Writer

```
while (TRUE) {
    rcu_read_lock {
        b' = b;
        x0 = *b';
        x1 = *(b'+1);
    }
    .. USE x0,x1 ..
}
```

```
while (TRUE) {
    .. PRODUCE y0,y1 ..
    nb = malloc(2);
    rcu_write_lock {
        *nb = y0; *(nb+1) = y1;
        barrier();
        ob = b; b = nb;
    } sync {
        free(ob); free(ob+1);
    }}
```

RCU Reader/Writer Optimized

```
while (TRUE) {
    rcu_read_lock {
        b' = b;
        x0 = *b';
        x1 = *(b'+1);
    }
    .. USE x0,x1 ..
}
```

nb = malloc(2);

```
while (TRUE) {
    .. PRODUCE y0,y1 ..
    nb = malloc(2);
    rcu_write_lock {
        *nb = y0; *(nb+1) = y1;
        barrier();
        ob = b; b = nb;
    } sync {
        nb = ob;
        free(ob+1);
    }}
```