Persistence Semantics for Weak Memory

Integrating Epoch Persistency with the TSO Memory Model

Azalea Raad Viktor Vafeiadis

Max Planck Institute for Software Systems (MPI-SWS)

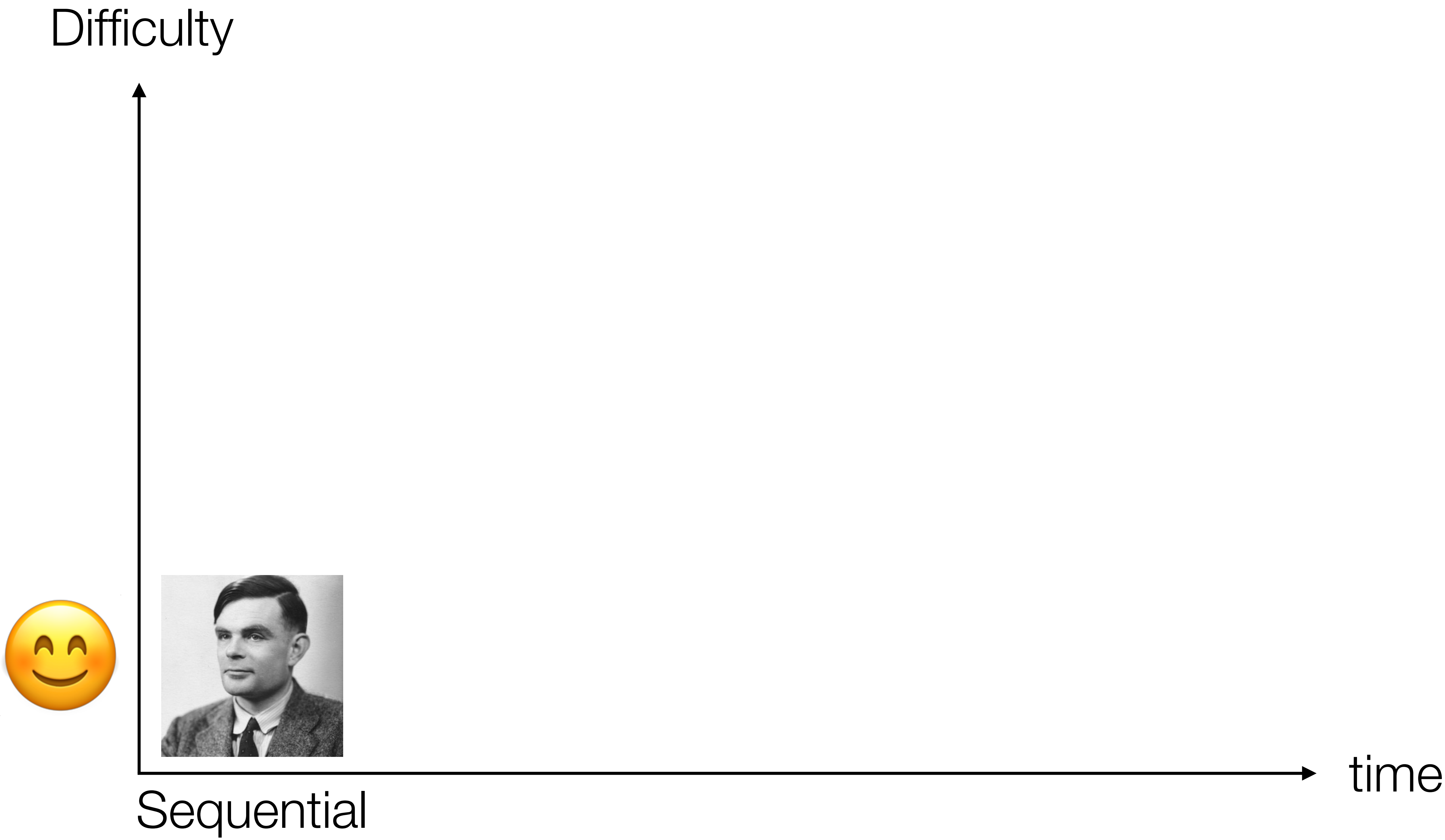
Thursday 8 November

OOPSLA 2018

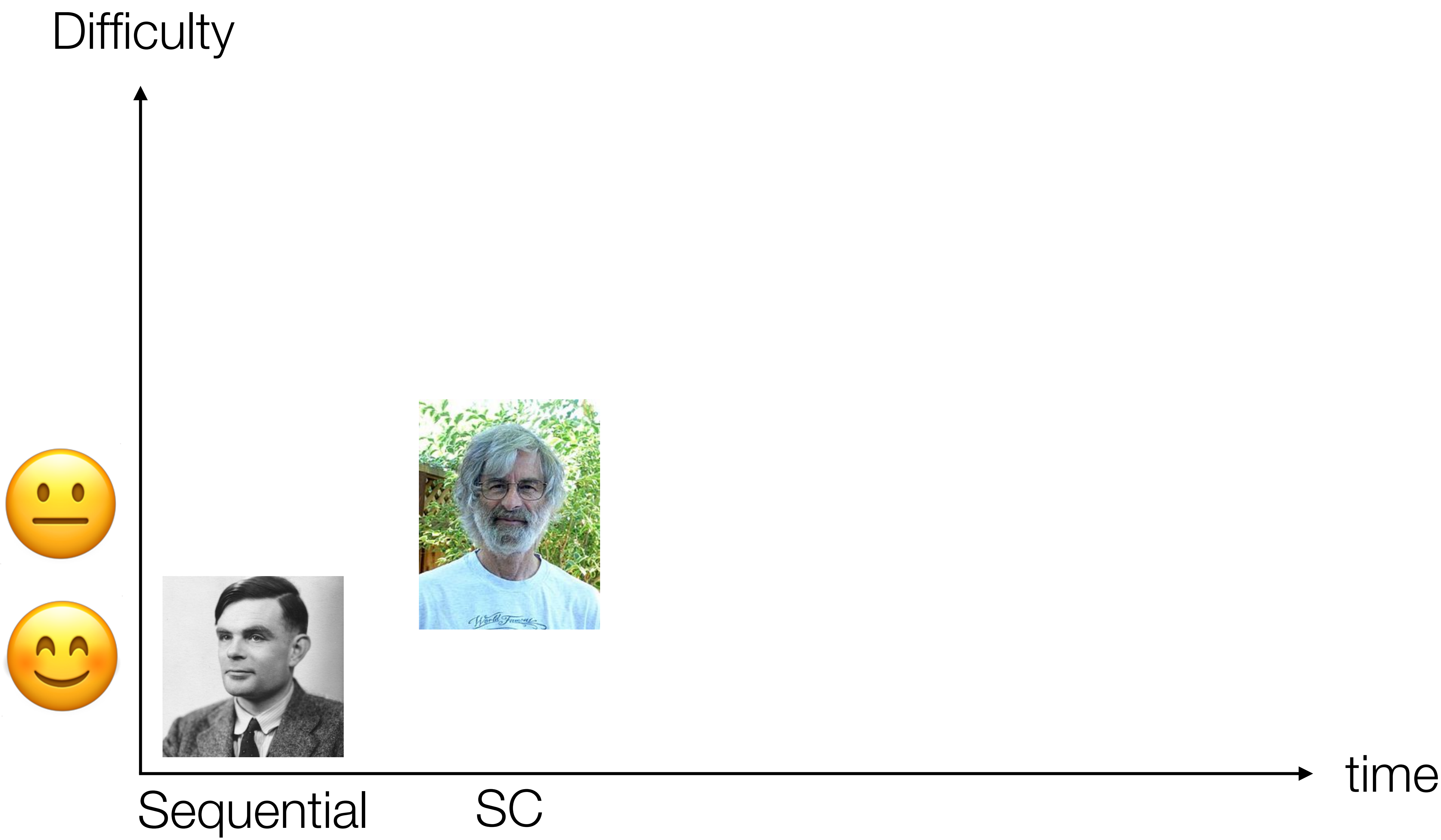
Boston, USA

azalea@mpi-sws.org SoundAndComplete.org @azalearaad

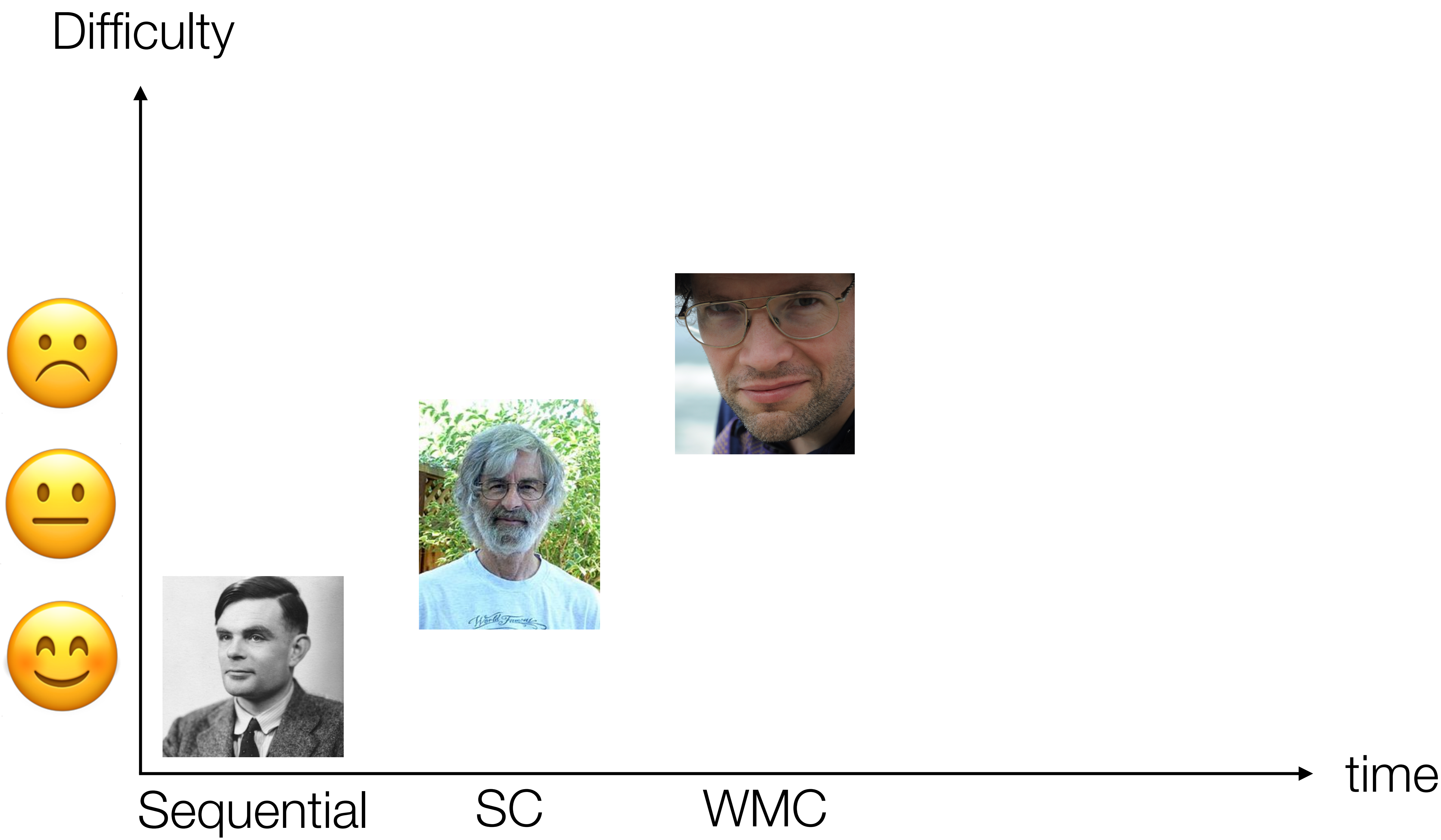
History



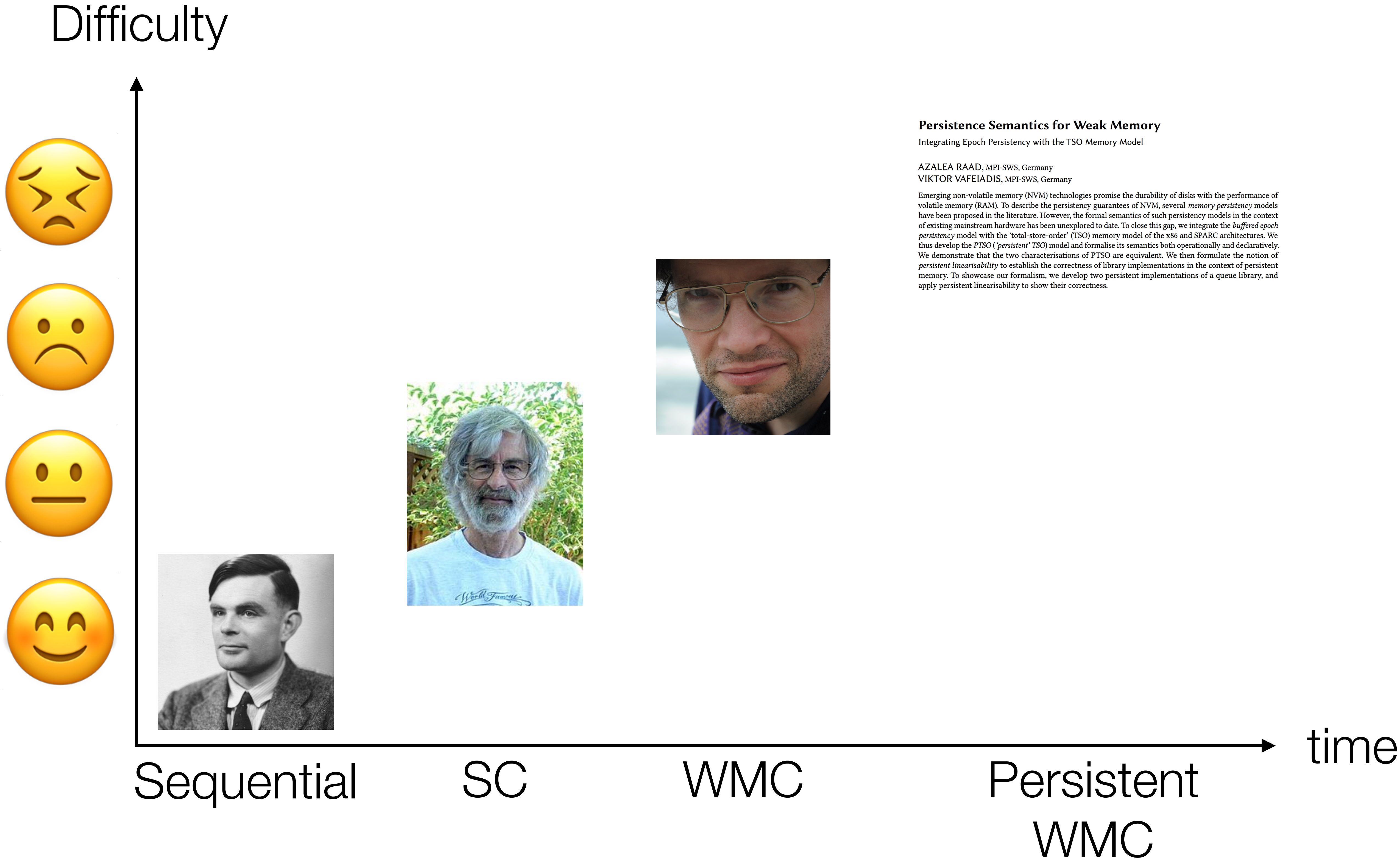
History



History



History



Persistence Semantics for Weak Memory

Integrating Epoch Persistency with the TSO Memory Model

AZALEA RAAD, MPI-SWS, Germany
VIKTOR VAFEIADIS, MPI-SWS, Germany

Emerging non-volatile memory (NVM) technologies promise the durability of disks with the performance of volatile memory (RAM). To describe the persistency guarantees of NVM, several *memory persistency* models have been proposed in the literature. However, the formal semantics of such persistency models in the context of existing mainstream hardware has been unexplored to date. To close this gap, we integrate the *buffered epoch persistency* model with the 'total-store-order' (TSO) memory model of the x86 and SPARC architectures. We thus develop the *PTSO* (*'persistent' TSO*) model and formalise its semantics both operationally and declaratively. We demonstrate that the two characterisations of PTSO are equivalent. We then formulate the notion of *persistent linearisability* to establish the correctness of library implementations in the context of persistent memory. To showcase our formalism, we develop two persistent implementations of a queue library, and apply persistent linearisability to show their correctness.

What is Persistent Memory?

Volatile memory

// $\texttt{x} = 0$

`x := 1`

// $\texttt{x} = 1$

// $\texttt{x} = \texttt{v}$: reading `x` yields `v`

What is Persistent Memory?

Volatile memory

// $x = 0$

`x := 1`

// $x = 1$

// no recovery

// $x = 0$

// $x = v$: reading x yields v

What is Persistent Memory?

Volatile memory

// $x = 0$

`x := 1`

// $x = 1$

// no recovery

// $x = 0$

Persistent memory

// $x = 0$

`x := 1`

// $x = 1$

// recovery routine

// $x = 0$ OR $x = 1$

// $x = v$: reading `x` yields `v`

What is Persistent Memory?

Volatile **memory**

// $x = 0$

`x := 1`

// $x = 1$

// no recovery

// $x = 0$

Persistent **memory**

// $x = 0$

`x := 1`

// $x = 1$

// recovery routine

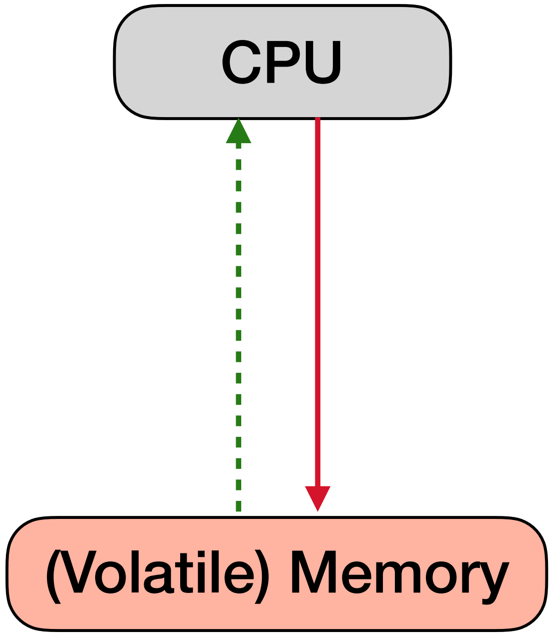
// $x = 0$ OR $x = 1$

persists are ***asynchronous*** (buffered): may not persist immediately

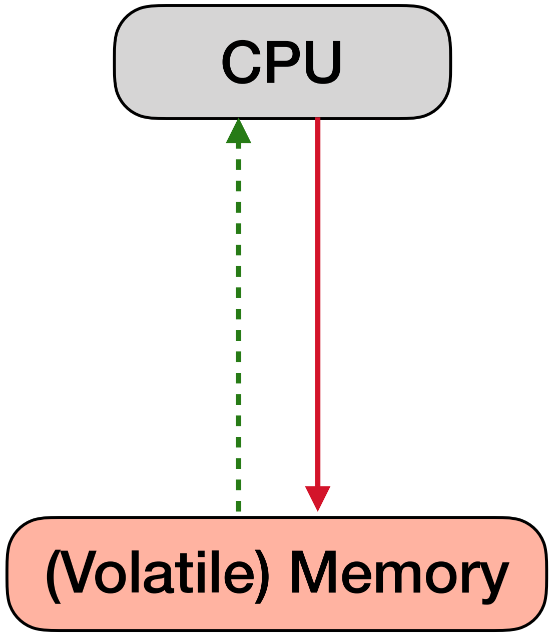
// $x = v$: reading x yields v

(Sequential) Hardware

(Sequential) Hardware

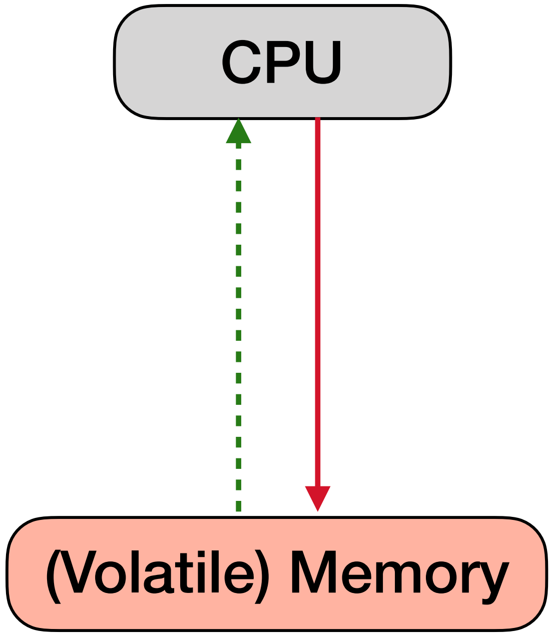


(Sequential) Hardware



`x:=1` : adds `x:=1` to memory

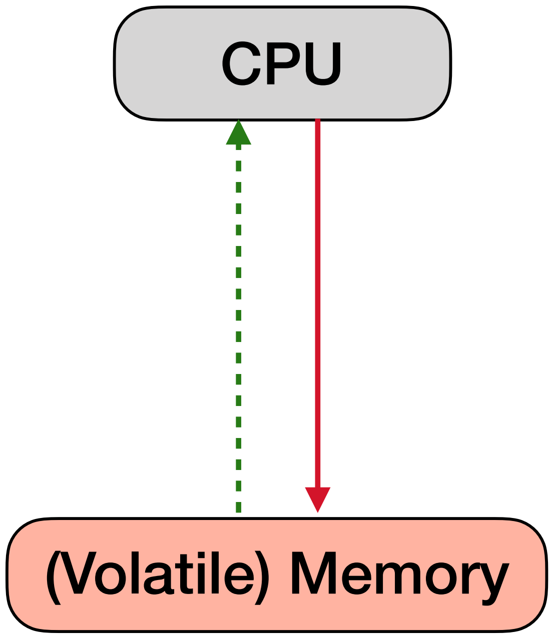
(Sequential) Hardware



`x:=1` : adds `x:=1` to memory

`a:=x` : reads `x` from memory

(Sequential) Hardware

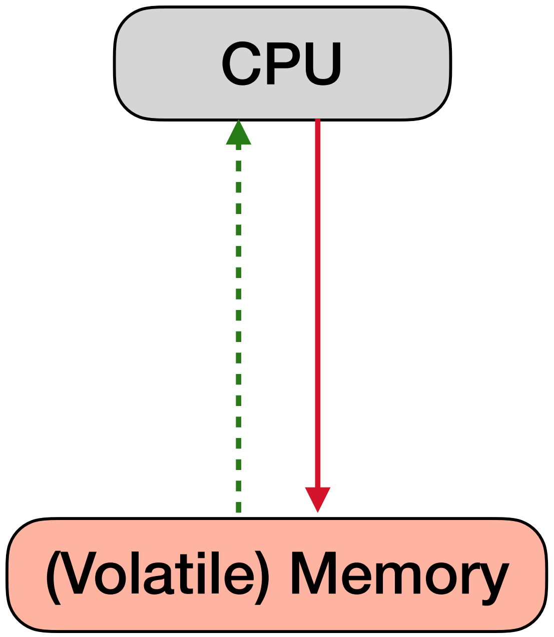


`x:=1` : adds `x:=1` to memory

`a:=x` : reads `x` from memory

memory lost

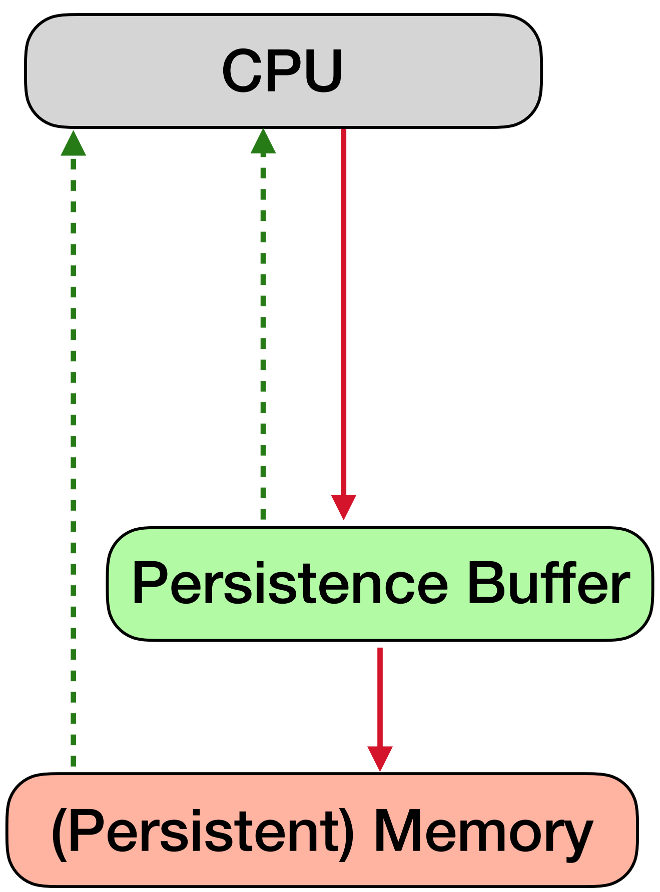
(Sequential) Hardware



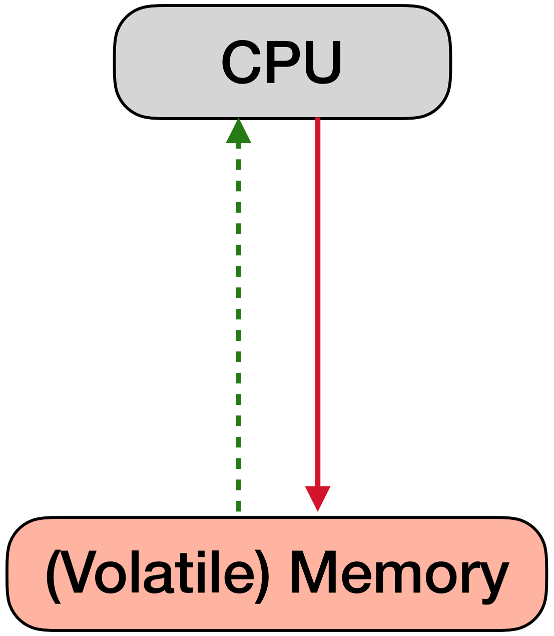
`x:=1` : adds `x:=1` to memory

`a:=x` : reads `x` from memory

memory lost



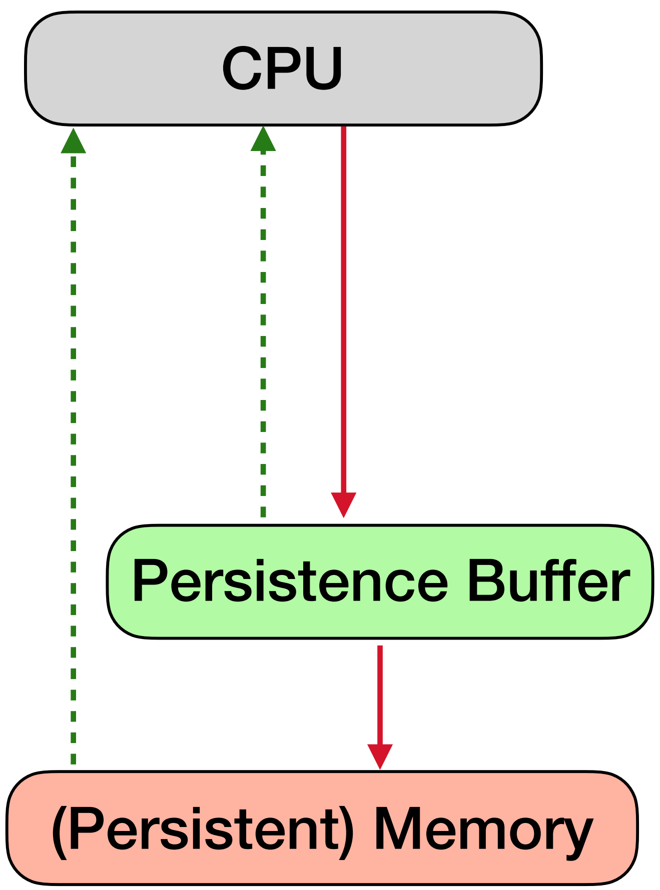
(Sequential) Hardware



`x:=1` : adds `x:=1` to memory

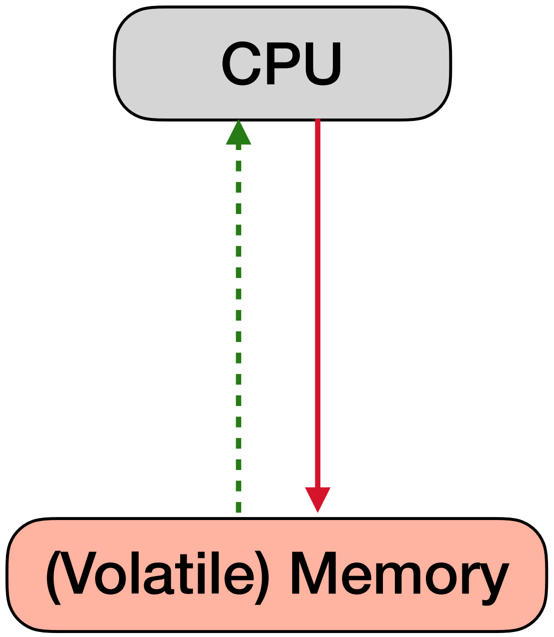
`a:=x` : reads `x` from memory

memory lost



`x:=1` : adds `x:=1` to p-buffer

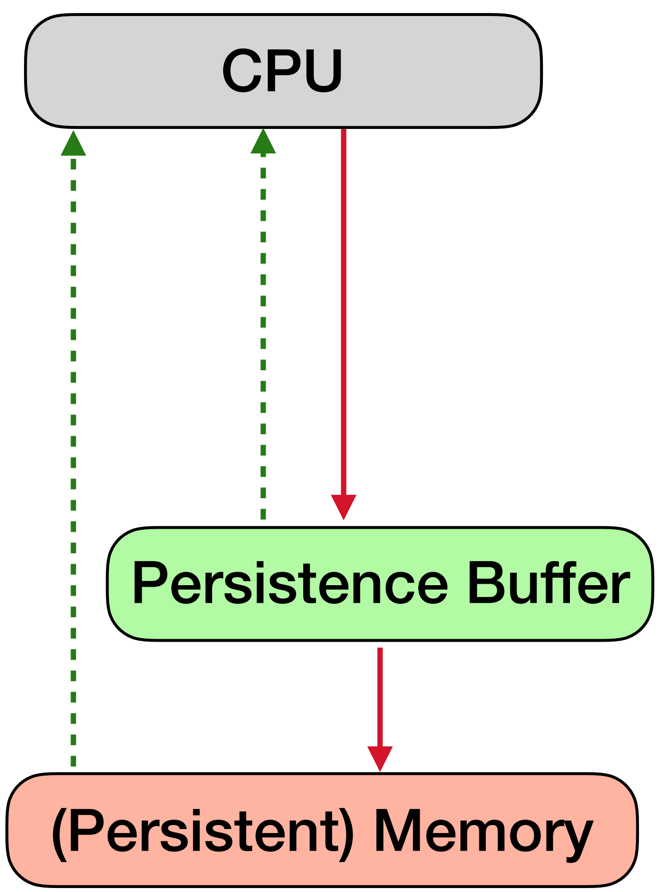
(Sequential) Hardware



`x:=1` : adds `x:=1` to memory

`a:=x` : reads `x` from memory

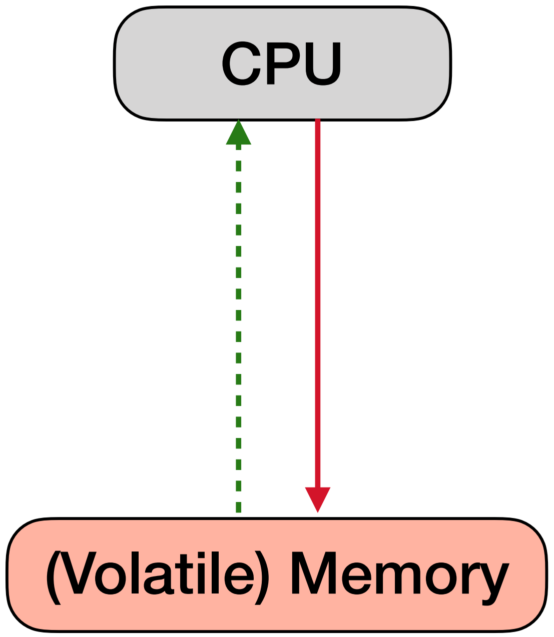
memory lost



`x:=1` : adds `x:=1` to p-buffer

`a:=x` : if p-buffer contains `x`, reads latest entry
else reads from memory

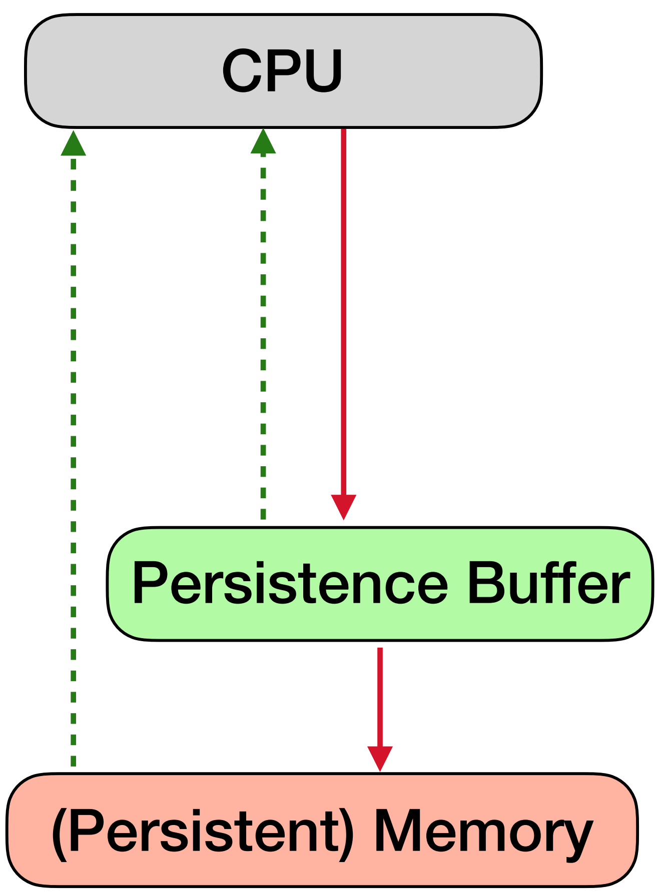
(Sequential) Hardware



`x:=1` : adds `x:=1` to memory

`a:=x` : reads `x` from memory

memory lost

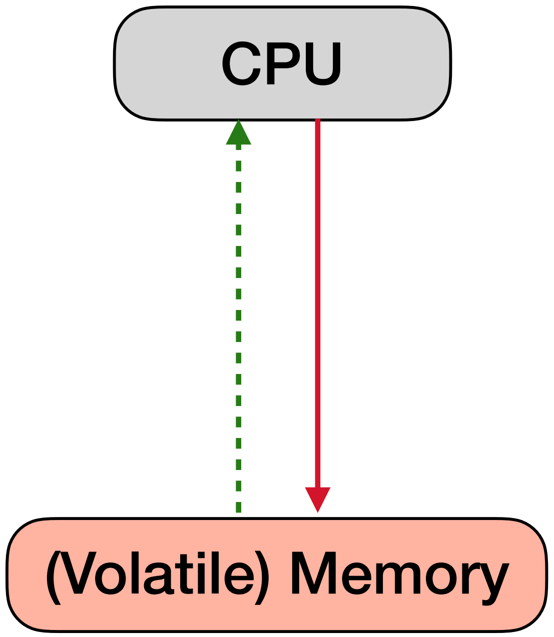


`x:=1` : adds `x:=1` to p-buffer

`a:=x` : if p-buffer contains `x`, reads latest entry
else reads from memory

p-buffer lost; memory ***retained***

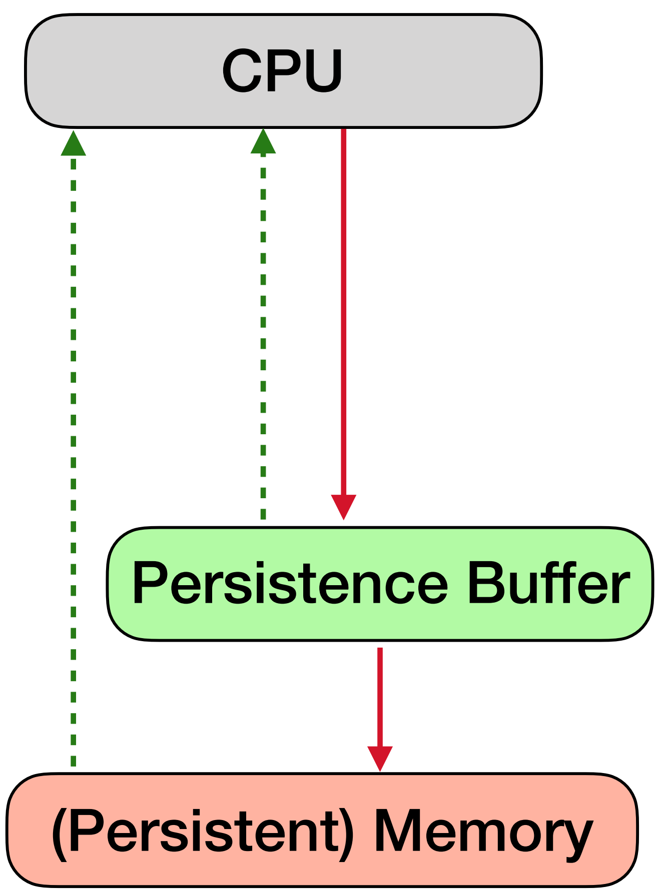
(Sequential) Hardware



`x:=1` : adds `x:=1` to memory

`a:=x` : reads `x` from memory

memory lost



`x:=1` : adds `x:=1` to p-buffer

`a:=x` : if p-buffer contains `x`, reads latest entry
else reads from memory

p-buffer lost; memory ***retained***

unbuffer* : p-buffer to memory

* at non-deterministic times

What is Memory Persistency Model?

- Memory ***consistency*** model describes:

  the order writes are made visible to other threads
  e.g. SC, TSO, …

What is Memory Persistency Model?

- Memory ***consistency*** model describes:

  the order writes are made visible to other threads
  e.g. SC, TSO, …

- Memory ***persistency*** model describes:

  the order writes are persisted to memory
  e.g. <u>Epoch Persistency</u>

What is Memory Persistency Model?

- Memory ***consistency*** model describes:

<u>Problem</u>

Formal
Epoch Persistency Model
for
Mainstream Hardware (***Weak*** Memory Models)

-
  the order writes are persisted to memory
  e.g. <u>Epoch Persistency</u>

What Can Go Wrong?

```
// x=0;y=0

x := 1;

y := 1;
```

```
// recovery routine

// x=0;y=0 OR x=1;y=1 OR x=1;y=0 OR x=0;y=1
```

What Can Go Wrong?

```
// x=0;y=0
x := 1;
y := 1;
```

// recovery routine

```
// x=0;y=0 OR x=1;y=1 OR x=1;y=0 OR x=0;y=1
```

!! Writes may persist out of order

What Can Go Wrong?

```
// x=0;y=0

x := 1;

y := 1;

// recovery routine

// x=0;y=0 OR x=1;y=1 OR x=1;y=0 OR x=0;y=1
```

!! Writes may persist out of order

☛ ***persistent fence*** `pfence`

Persistent Fence

```
// x=0;y=0
x := 1;
pfence;
y := 1;
```

// recovery routine

// x=0;y=0 OR x=1;y=1 OR x=1;y=0 ~~OR x=0;y=1~~

Persistent Fence

```
a  x := 1;
b  y := 2;
c  x := 3;
   pfence;
d  z := 4;
```

Persistent Fence

- writes on ***same locations*** persist in <u>execution order</u>

```
a   x := 1;
b   y := 2;
c   x := 3;
    pfence;
d   z := 4;
```

a persists before c

Persistent Fence

- writes on ***same locations*** persist in execution order
- writes on ***different locations*** are unordered

```
a  x := 1;
b  y := 2;
c  x := 3;
   pfence;
d  z := 4;
```

a persists before c

a b may persist in any order

Persistent Fence

- writes on ***same locations*** persist in <u>execution order</u>
- writes on ***different locations*** are <u>unordered</u>
- `pfence` adds a new ***epoch***

```
a   x := 1;  ┐
b   y := 2;  │ epoch 1
c   x := 3;  ┘
    pfence;
d   z := 4;  ] epoch 2
```

a persists before c

a b may persist in any order

Persistent Fence

- writes on ***same locations*** persist in <u>execution order</u>
- writes on ***different locations*** are <u>unordered</u>
- `pfence` adds a new ***epoch***
- writes persist in <u>epoch order</u>

```
a   x := 1;
b   y := 2;    epoch 1
c   x := 3;
    pfence;
d   z := 4;    epoch 2
```

a persists before c

a b may persist in any order

a b c persist before d

What Can Go Wrong (Continued)?

```
// x=0;y=0
x := 1;
pfence;
y := 1;
```

asynchronous (buffered) →

// recovery routine

`// x=0;y=0 OR x=1;y=1 OR x=1;y=0`

!! Execution continues ahead of persistence

What Can Go Wrong (Continued)?

```
// x=0;y=0
 x := 1;
 pfence;
 y := 1;
```

asynchronous (buffered) → pfence

// recovery routine

// $\boxed{\texttt{x=0;y=0}}$ OR x=1;y=1 OR x=1;y=0

‼ Execution continues ahead of persistence

☛ ***persistent sync*** `psync`

What Can Go Wrong (Continued)?

```
// x=0;y=0
x := 1;
pfence;
y := 1;
```

asynchronous (buffered) → pfence

// recovery routine

```
// x=0;y=0 OR x=1;y=1 OR x=1;y=0
```

!! Execution continues ahead of persistence

☛ ***persistent sync*** `psync`

`C1; psync; C2`

- same persist-ordering as `pfence`
- `C2` executed only when ***all*** `C1` writes have persisted

Persistent Sync

```
// x=0;y=0
x := 1;
psync;
y := 1;
```

// recovery routine

// ~~x=0;y=0 OR~~ x=1;y=1 OR x=1;y=0

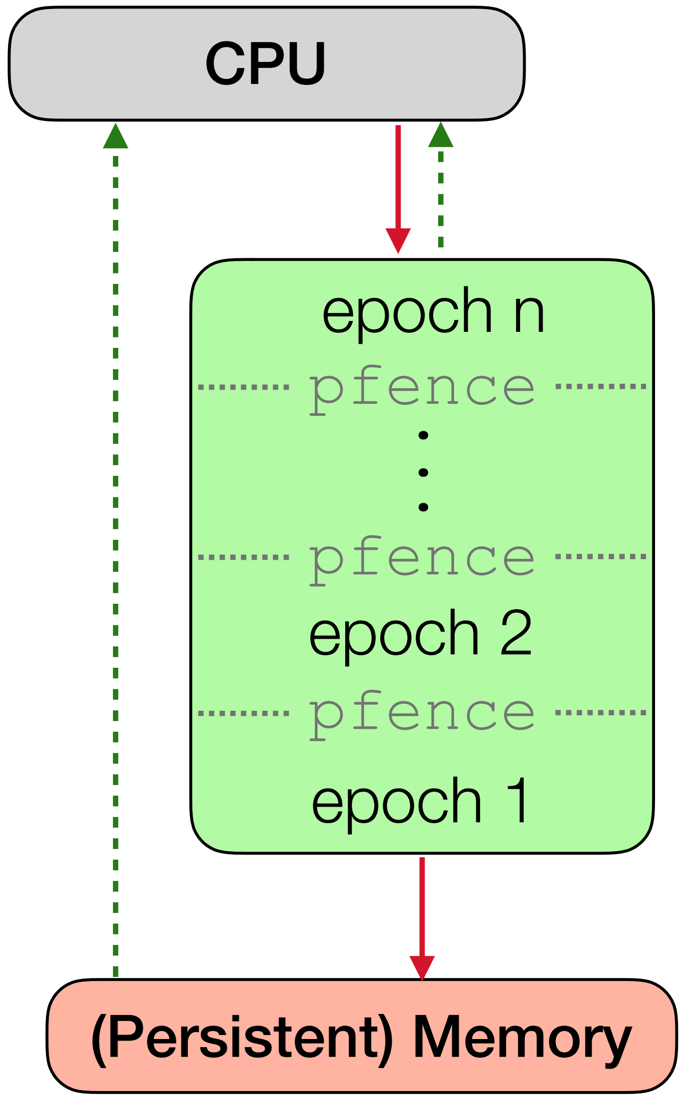
!! Execution continues ahead of persistence

☛ ***persistent sync*** `psync`

`C1; psync; C2`

- same persist-ordering as `pfence`
- `C2` executed only when ***all*** `C1` writes have persisted

(Sequential) Hardware

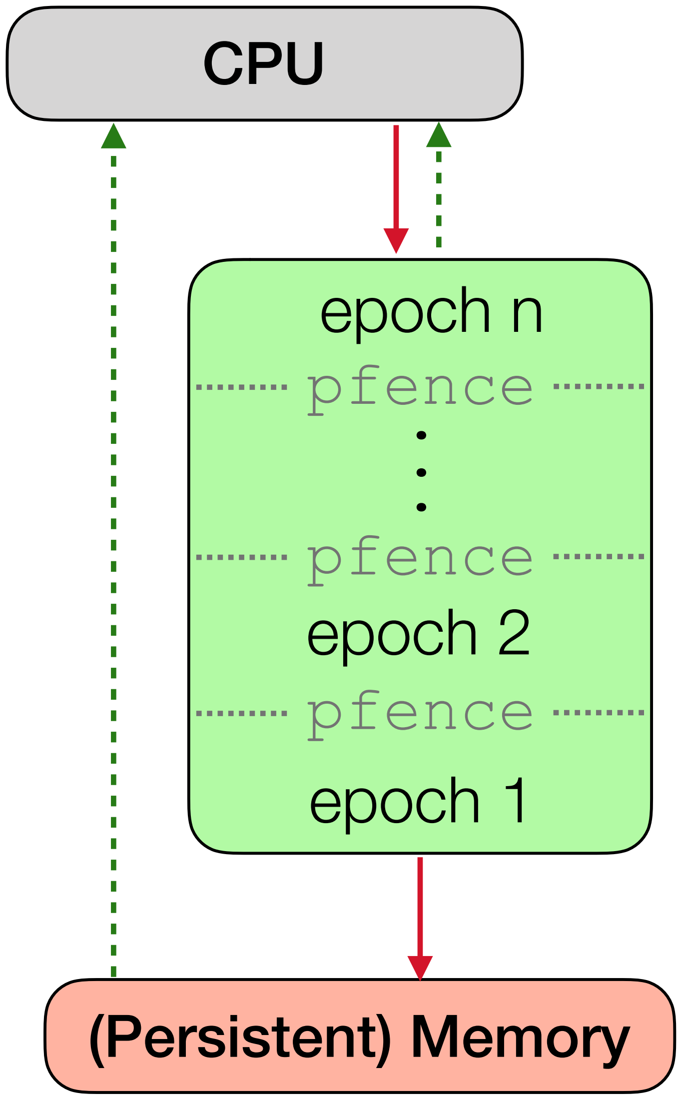


`x:=1` : adds `x:=1` to p-buffer

`a:=x` : if p-buffer contains `x`, reads latest entry
else reads from memory

p-buffer lost; memory retained

(Sequential) Hardware



`x:=1` : adds `x:=1` to p-buffer

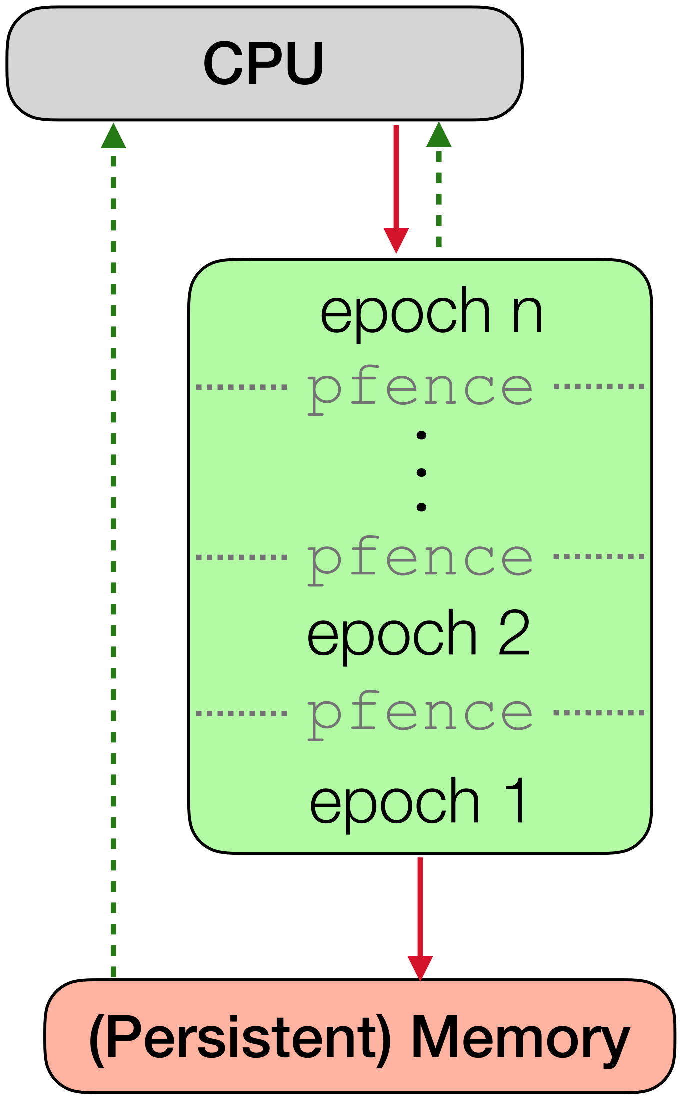
`a:=x` : if p-buffer contains `x`, reads latest entry else reads from memory

p-buffer lost; memory retained

unbuffer* : p-buffer to memory (in epoch order)

* at non-deterministic times

(Sequential) Hardware



`x:=1` : adds `x:=1` to p-buffer

`a:=x` : if p-buffer contains `x`, reads latest entry else reads from memory

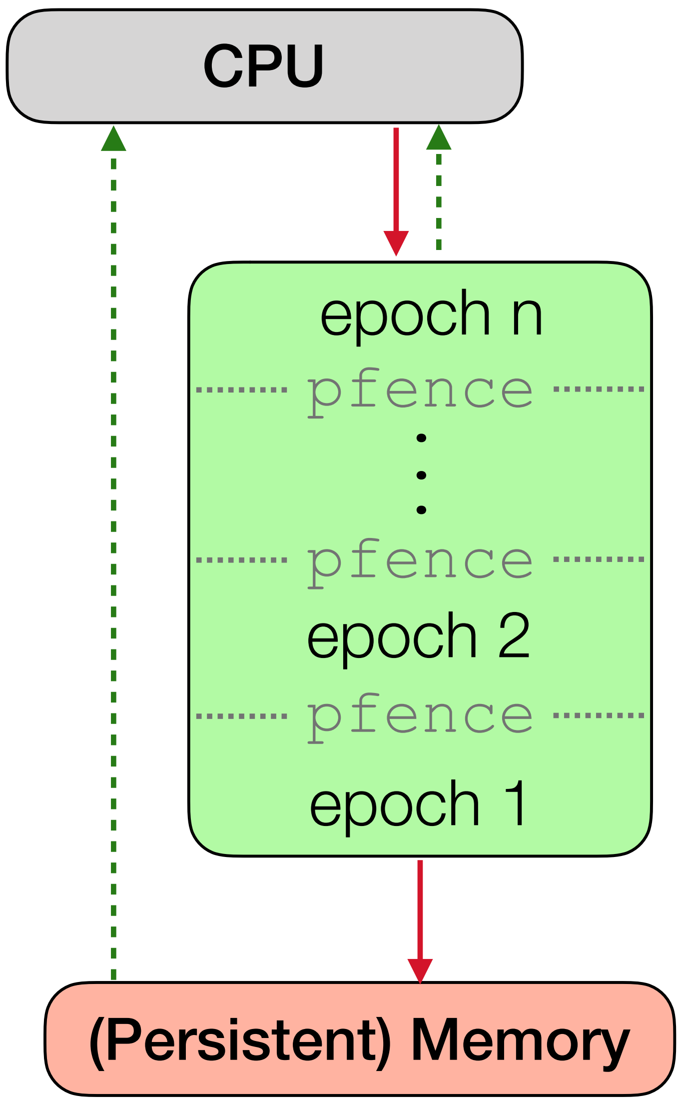
p-buffer lost; memory retained

unbuffer* : p-buffer to memory (in epoch order)

`pfence` : introduces a new epoch in p-buffer

* at non-deterministic times

(Sequential) Hardware



`x:=1` : adds `x:=1` to p-buffer

`a:=x` : if p-buffer contains `x`, reads latest entry
else reads from memory

p-buffer lost; memory retained

unbuffer* : p-buffer to memory (in epoch order)

`pfence` : introduces a new epoch in p-buffer

`psync` : flushes the entire p-buffer to memory

* at non-deterministic times

What about Concurrency?

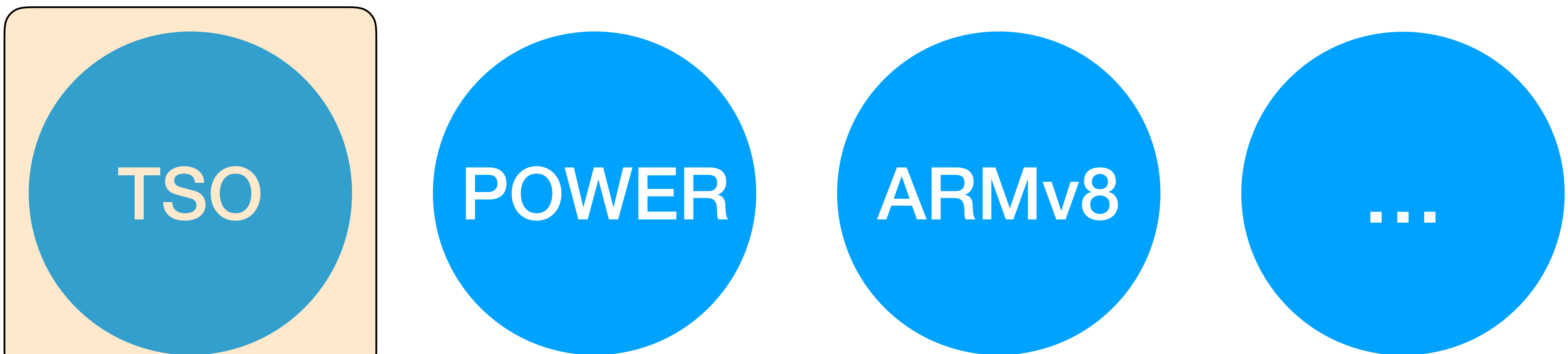
TSO

POWER

ARMv8

...

What about Concurrency?



Contributions

Contributions

- PTSO: <u>First</u> formal epoch persistency semantics under mainstream hardware
  - ***Operational*** model
  - ***Declarative*** model
  - ***Equivalence*** of the two models

Contributions

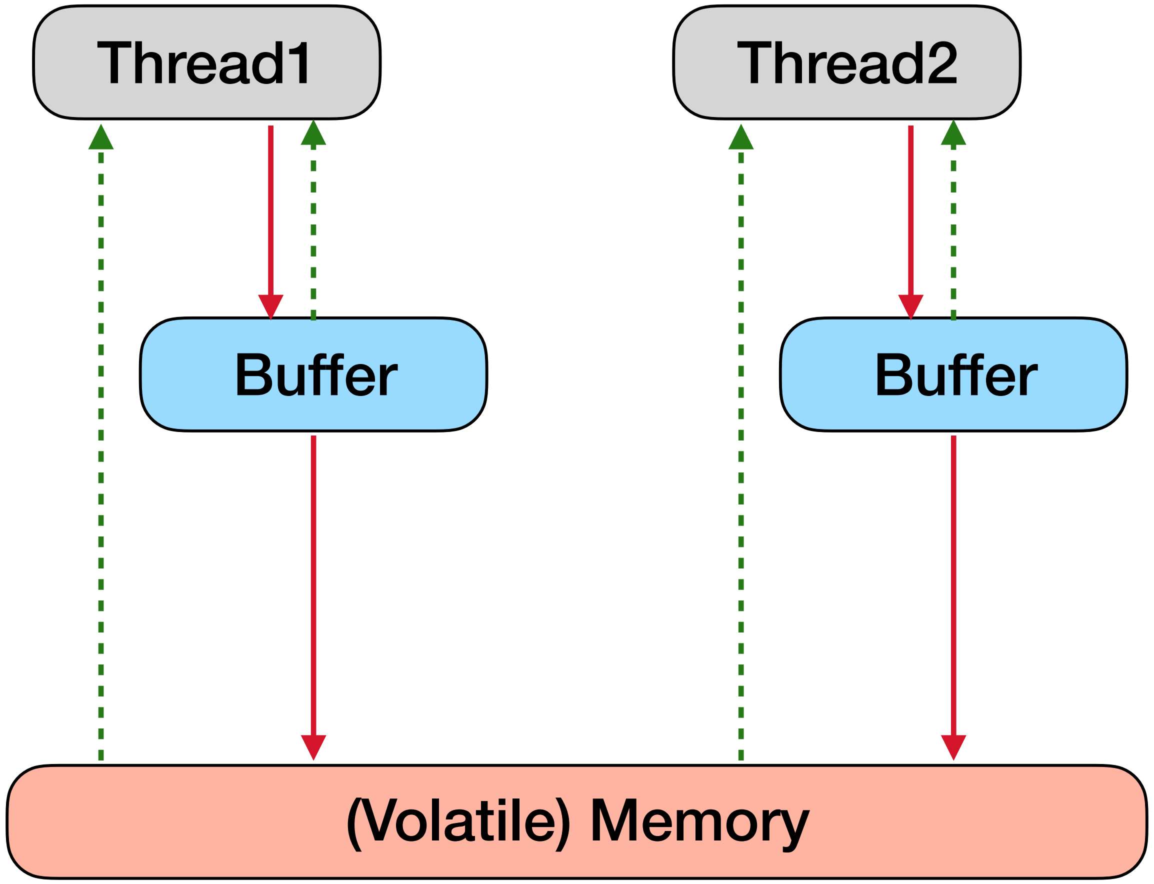
- PTSO: <u>First</u> formal epoch persistency semantics under mainstream hardware
  - ***Operational*** model
  - ***Declarative*** model
  - ***Equivalence*** of the two models

Contributions

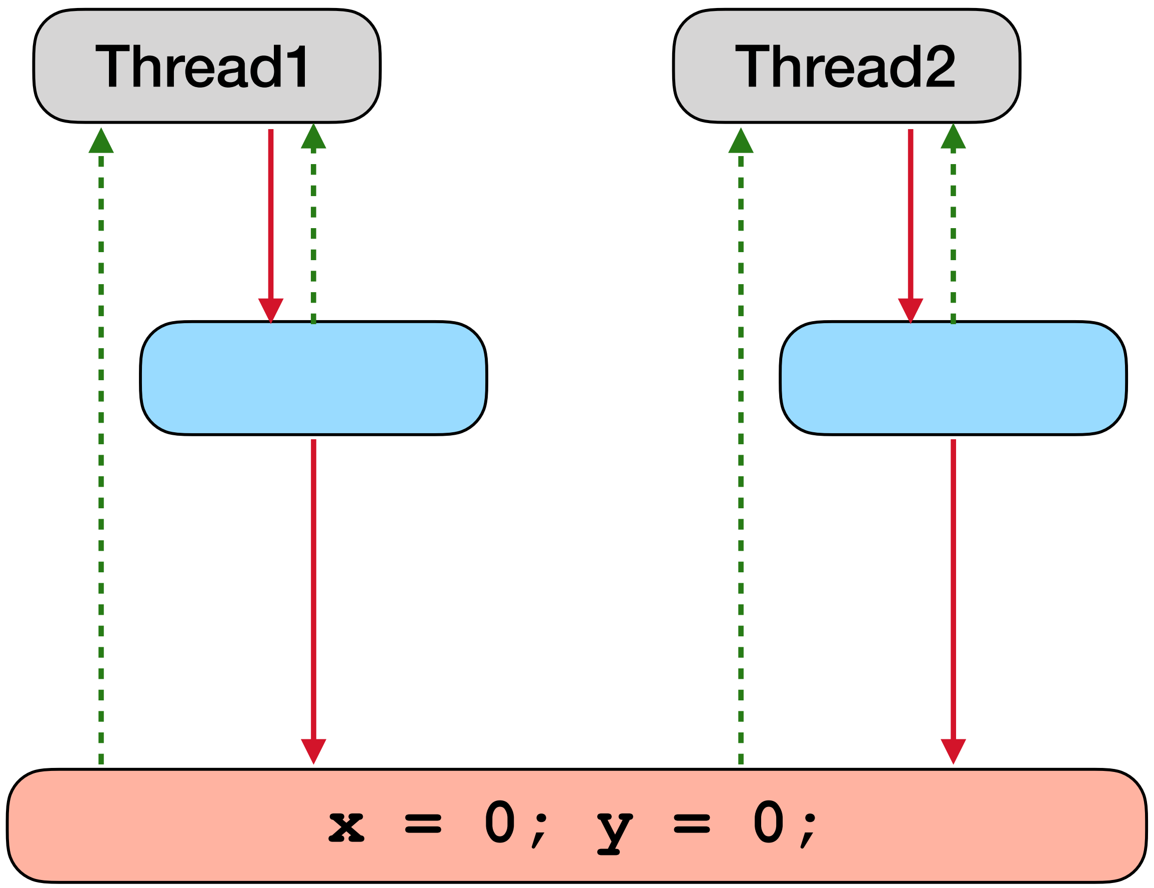
- PTSO: First formal epoch persistency semantics under mainstream hardware
  - ***Operational*** model
  - ***Declarative*** model
  - ***Equivalence*** of the two models
- Verifying programs under PTSO
  - PTSO programming ***pattern***
  - Correctness condition: ***persistent linearisability***
  - Verified several ***examples*** under PTSO

Total Store Ordering (TSO)

Total Store Ordering (TSO)



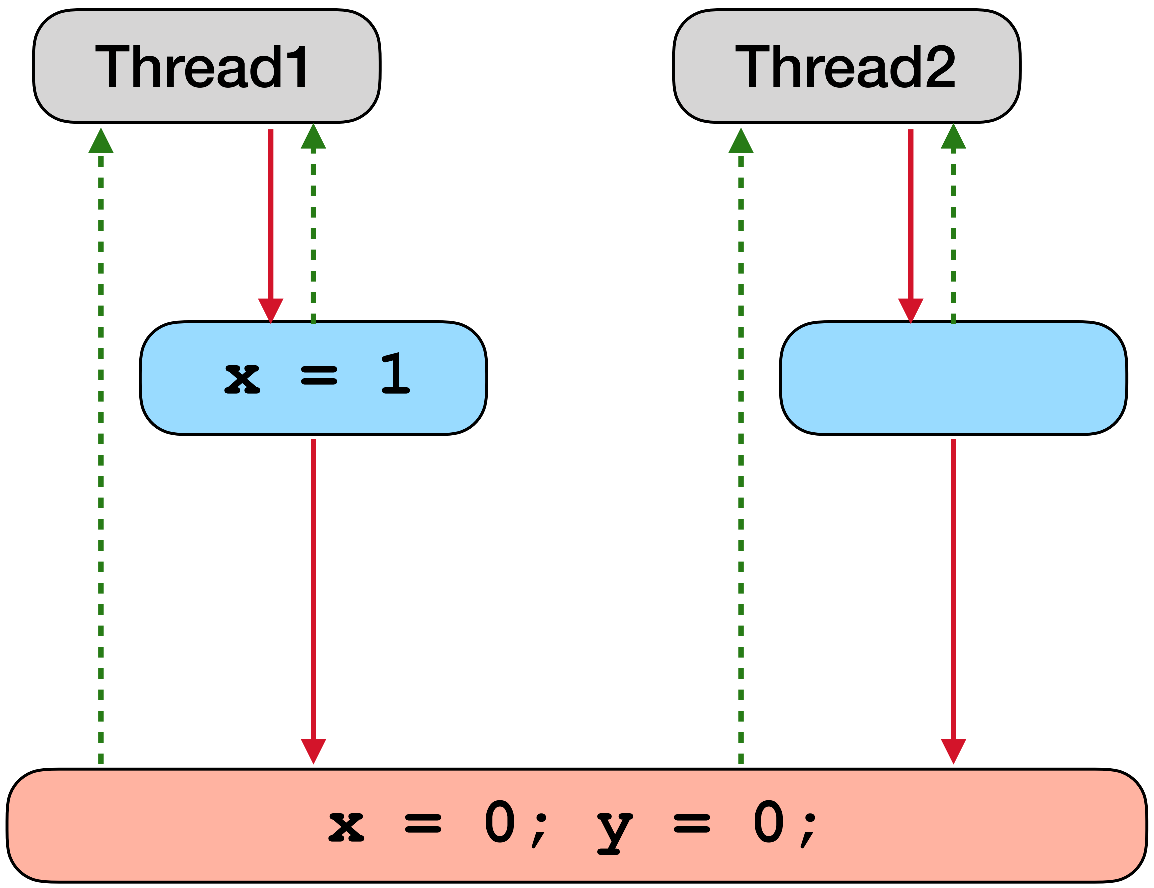
Total Store Ordering (TSO)



Thread1	Thread2
`x := 1;`	`y := 1;`
`a := y;`	`c := x;`

Store Buffering (SB)

Total Store Ordering (TSO)



Thread1

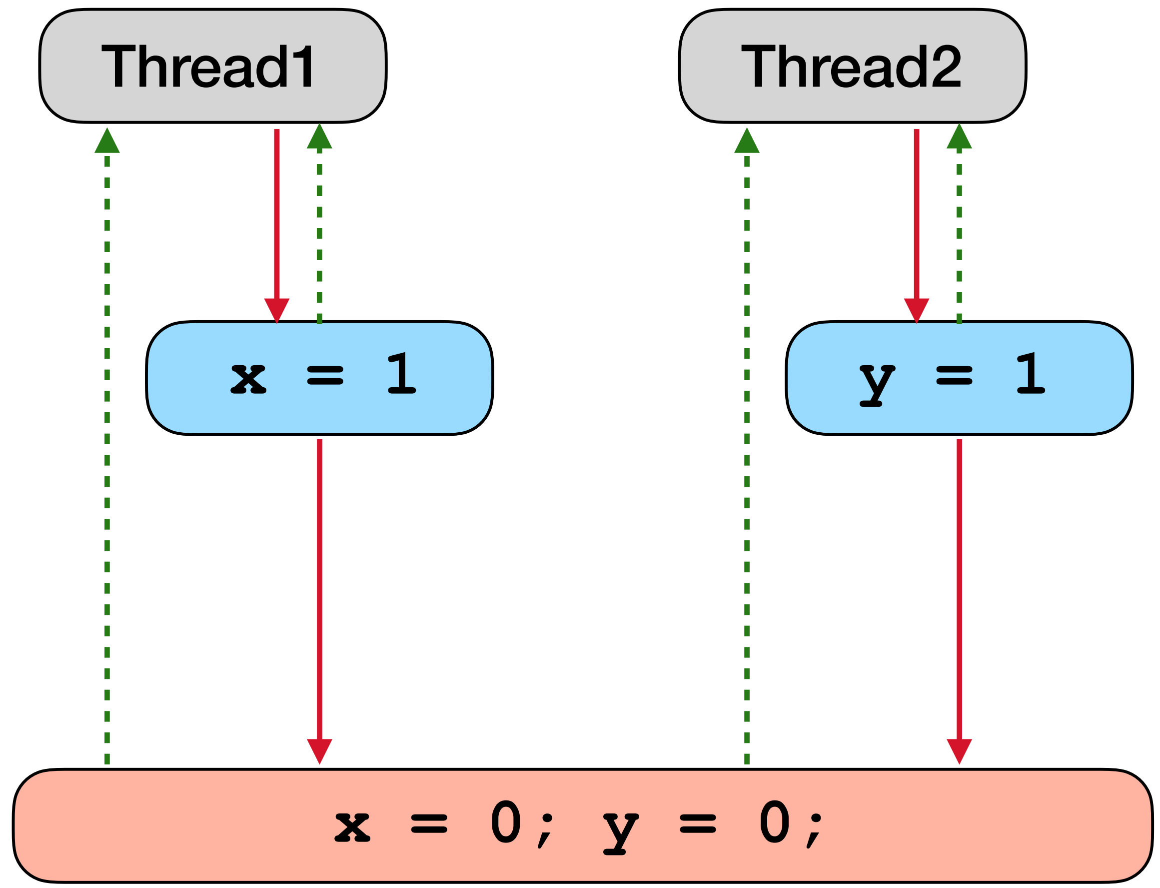
```
x := 1;
a := y;
```

Thread2

```
y := 1;
c := x;
```

Store Buffering (SB)

Total Store Ordering (TSO)



<u>Thread1</u>

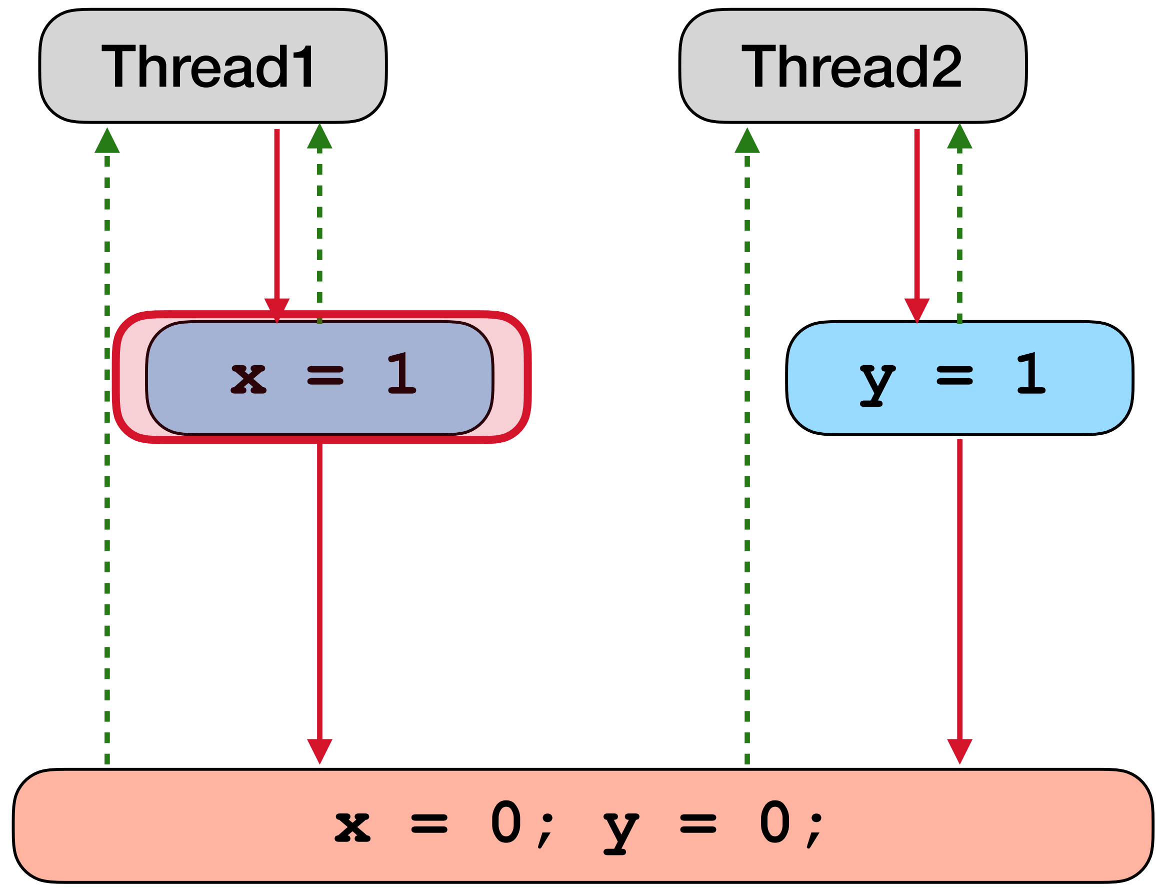
```
x := 1;
a := y;
```

<u>Thread2</u>

```
y := 1;
c := x;
```

Store Buffering (SB)

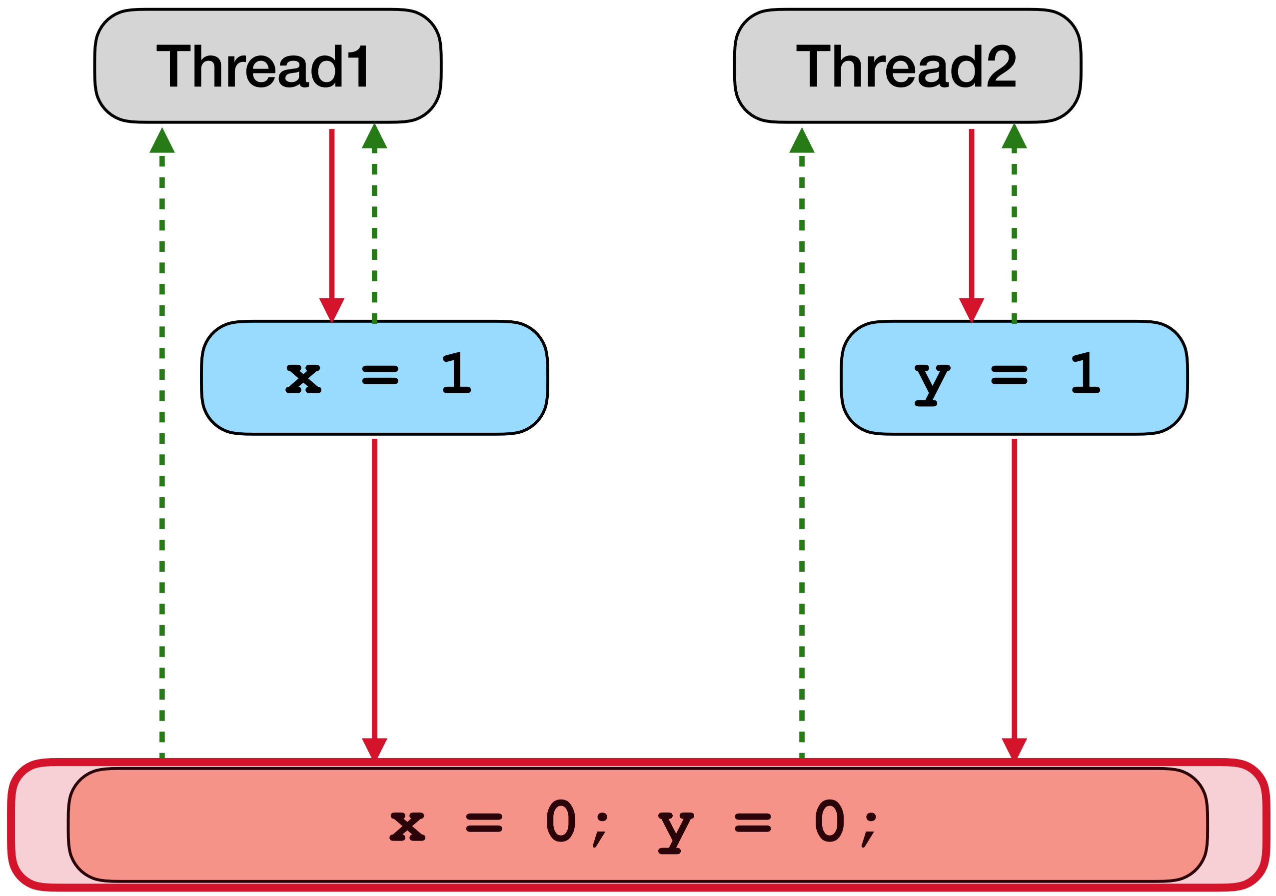
Total Store Ordering (TSO)



<u>Thread1</u>	<u>Thread2</u>
`x := 1;`	`y := 1;`
`a := y;`	`c := x;`

Store Buffering (SB)

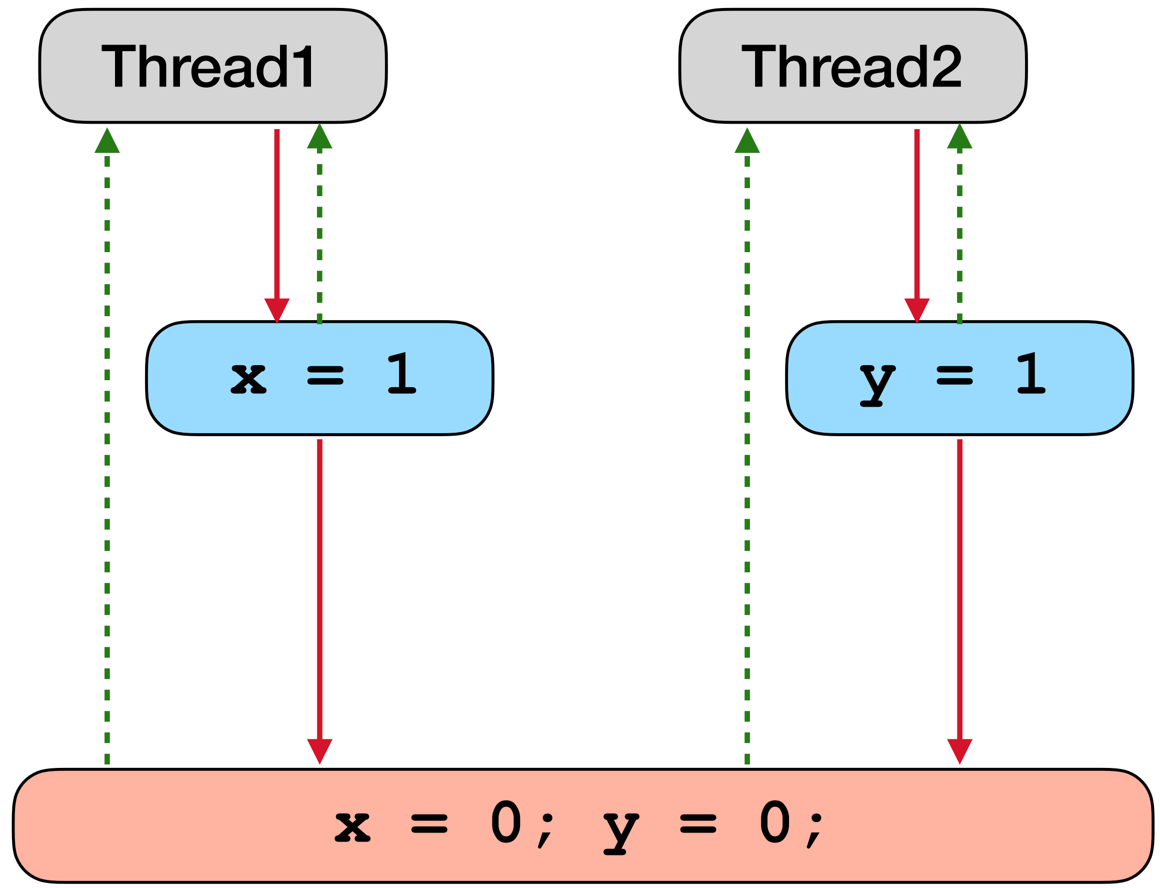
Total Store Ordering (TSO)



Thread1	Thread2
`x := 1;`	`y := 1;`
`a := y;`	`c := x;`

Store Buffering (SB)

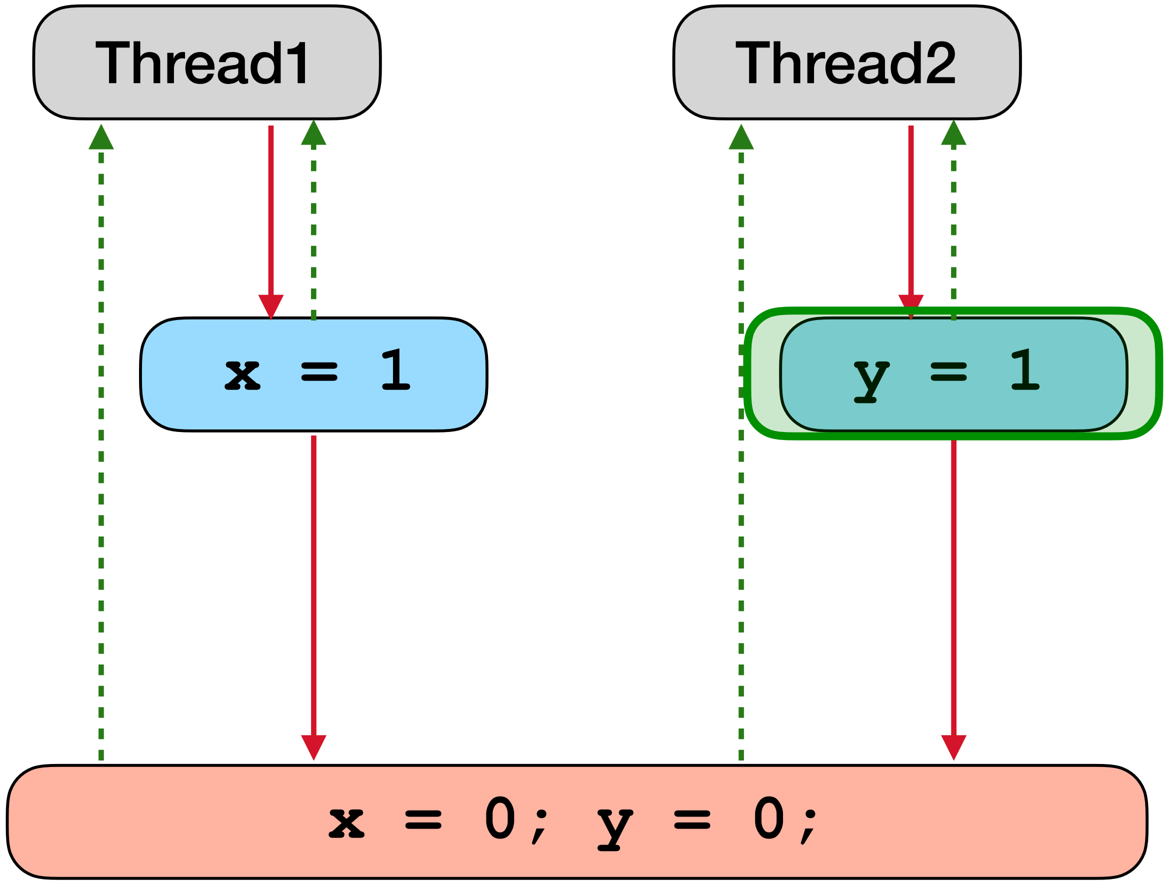
Total Store Ordering (TSO)

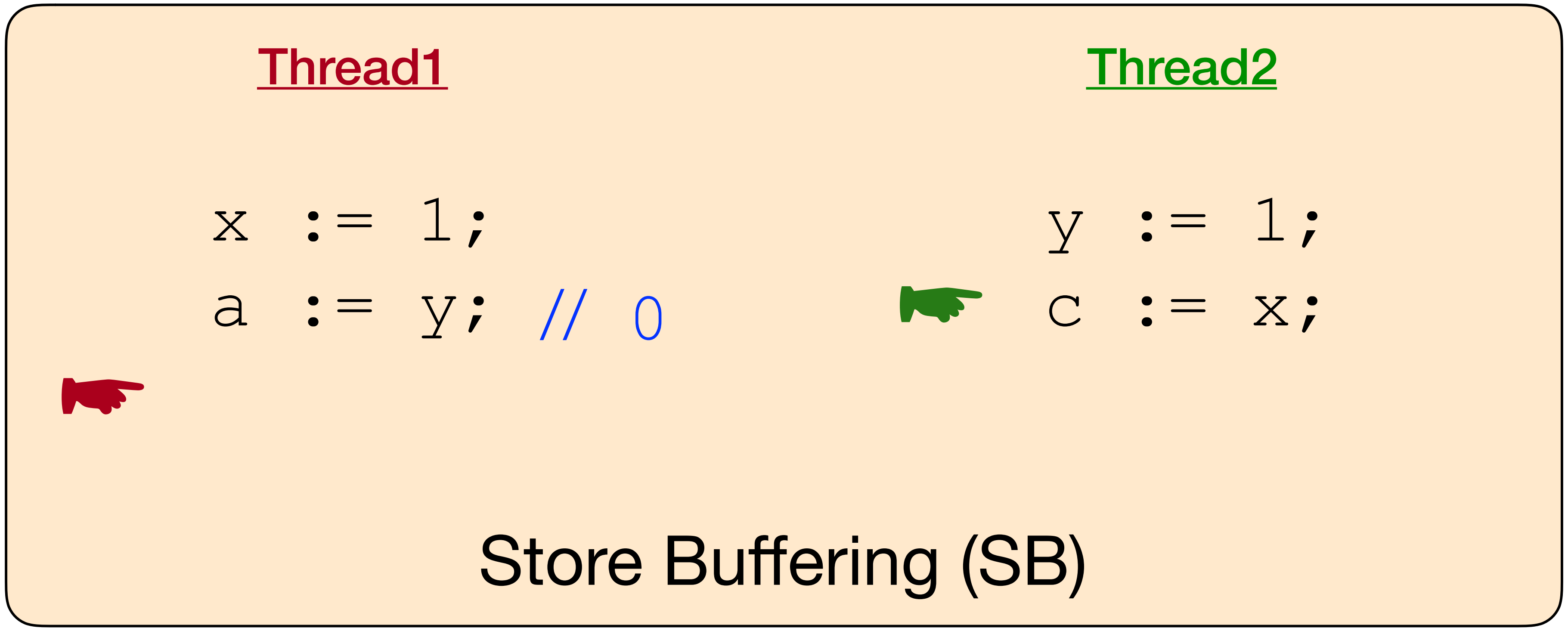


<u>Thread1</u>	<u>Thread2</u>
`x := 1;`	`y := 1;`
`a := y; // 0`	`c := x;`

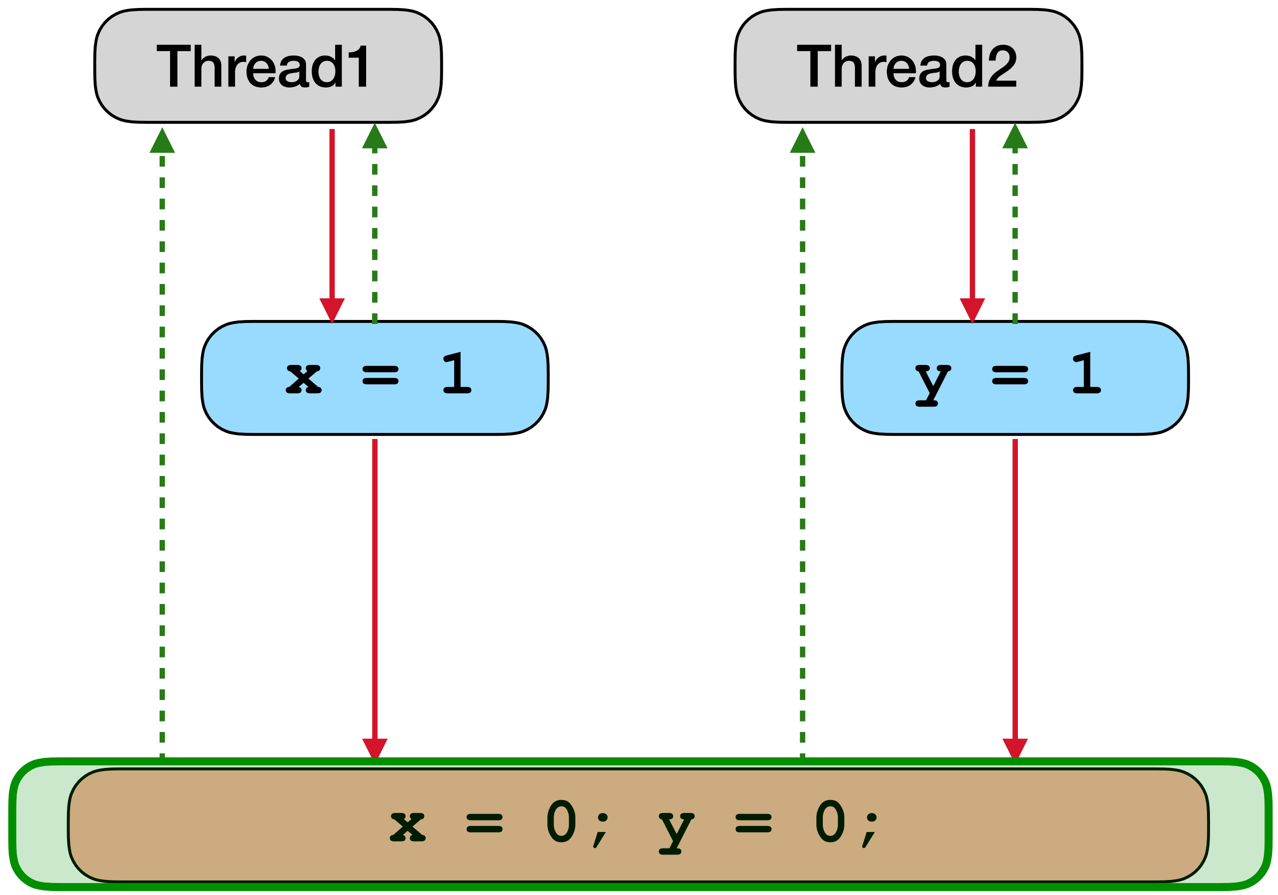
Store Buffering (SB)

Total Store Ordering (TSO)





Total Store Ordering (TSO)



Thread1

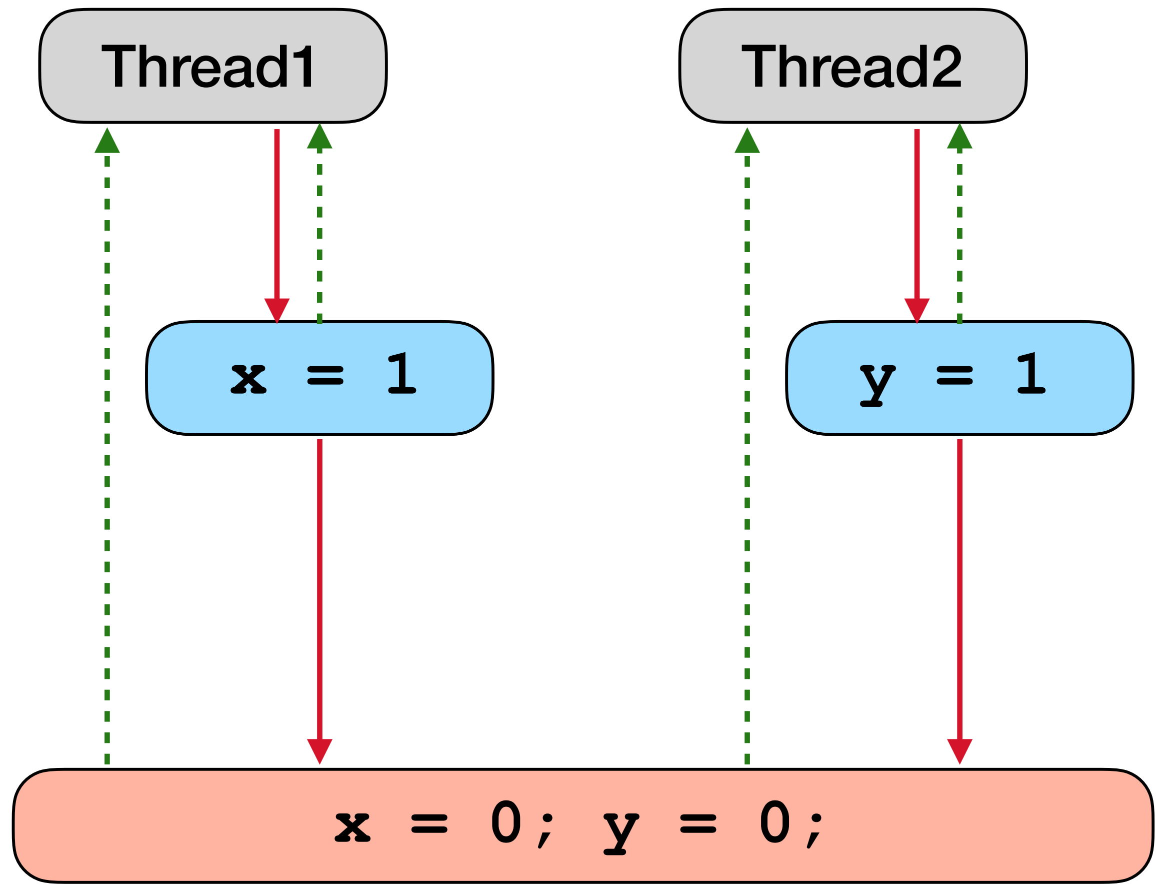
```
x := 1;
a := y; // 0
```

Thread2

```
y := 1;
c := x;
```

Store Buffering (SB)

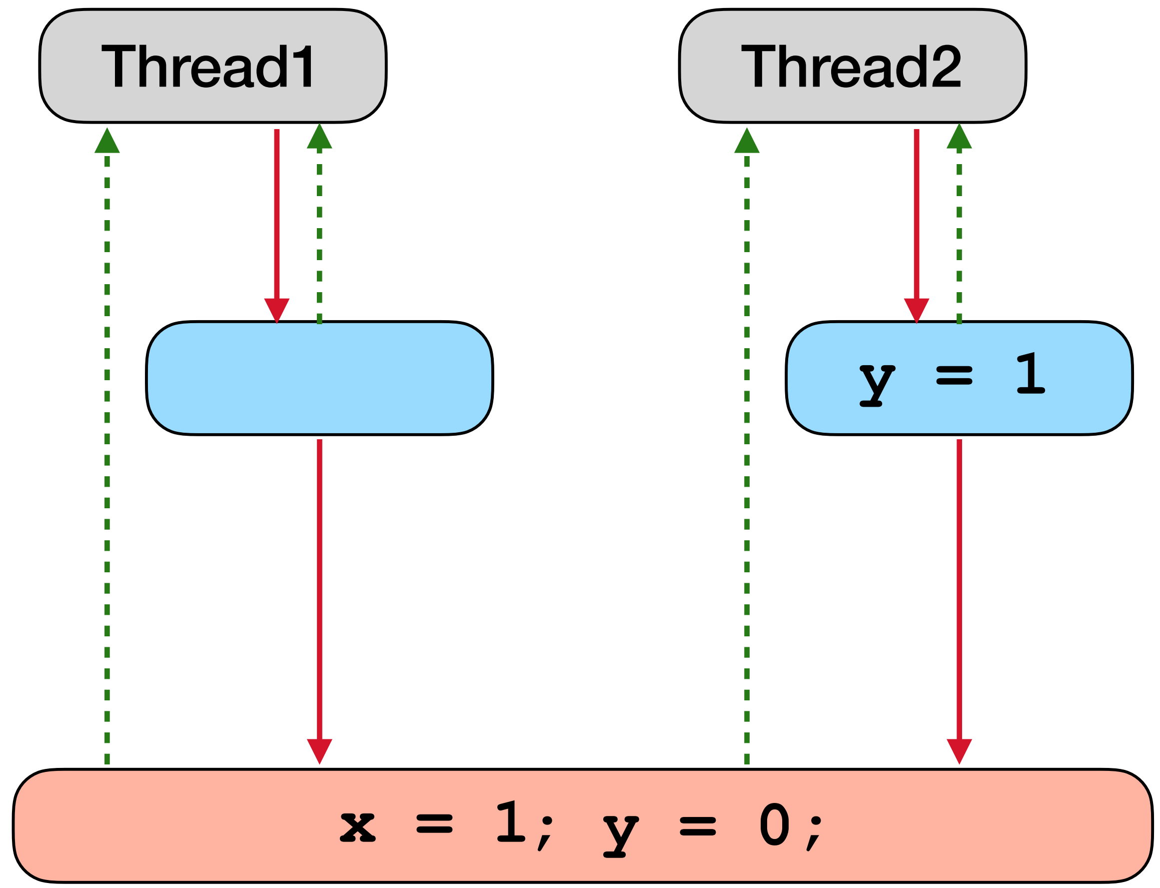
Total Store Ordering (TSO)



Thread1	Thread2
`x := 1;`	`y := 1;`
`a := y; // 0`	`c := x; // 0`

Store Buffering (SB)

Total Store Ordering (TSO)



<u>Thread1</u>

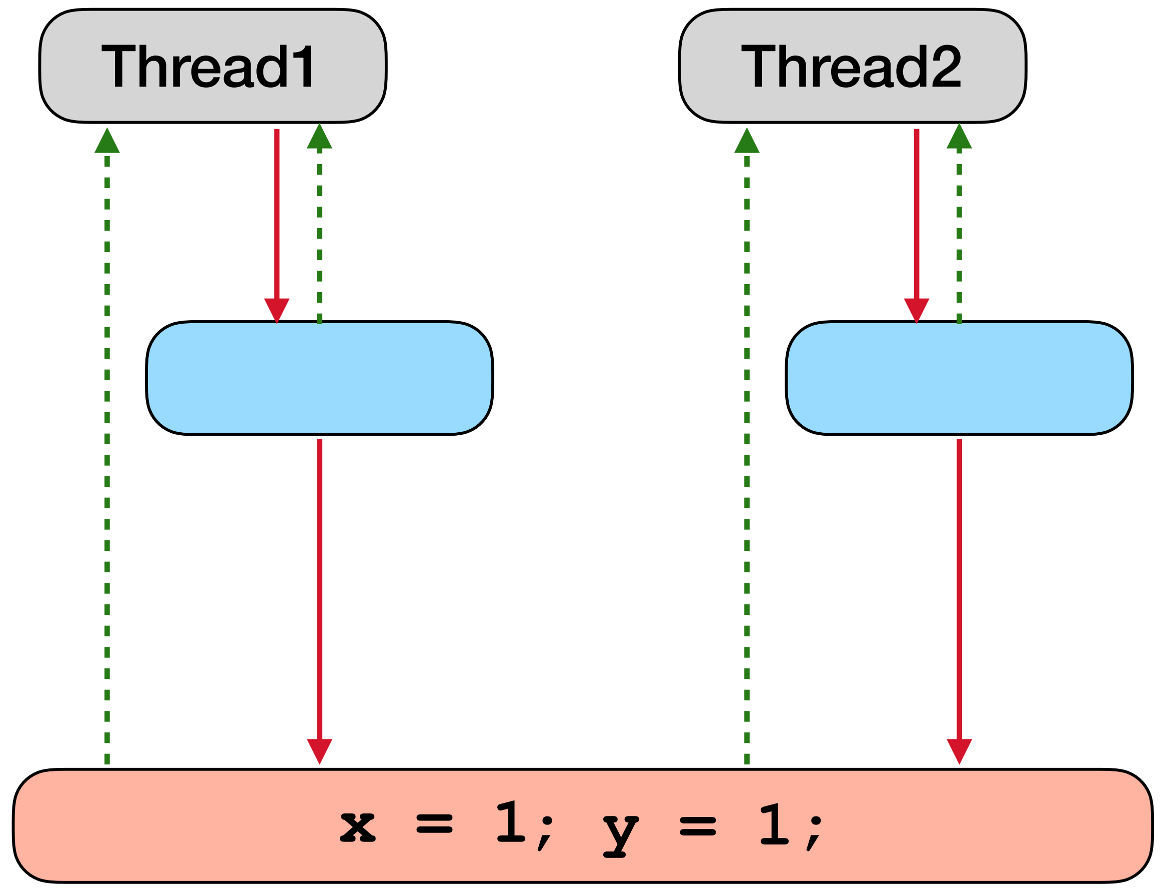
```
x := 1;
a := y; // 0
```

<u>Thread2</u>

```
y := 1;
c := x; // 0
```

Store Buffering (SB)

Total Store Ordering (TSO)



Thread1

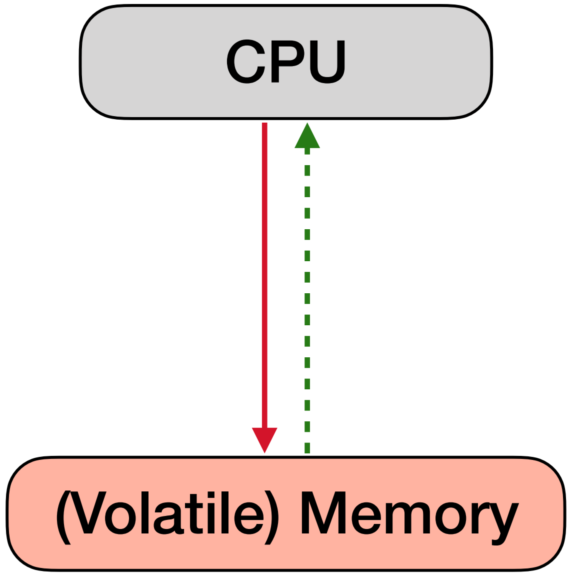
```
x := 1;
a := y; // 0
```

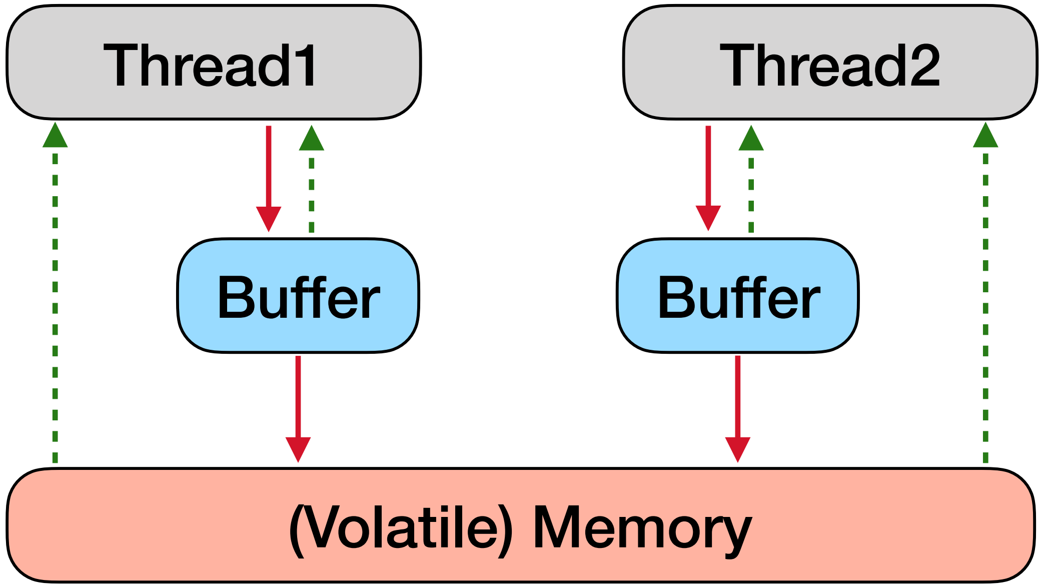
Thread2

```
y := 1;
c := x; // 0
```

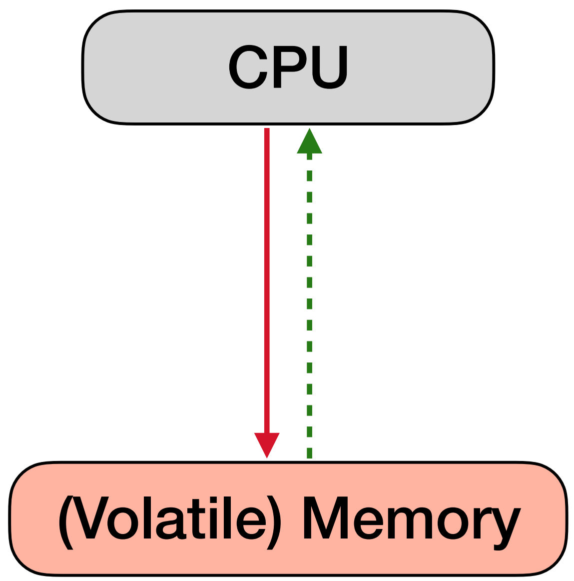
Store Buffering (SB)

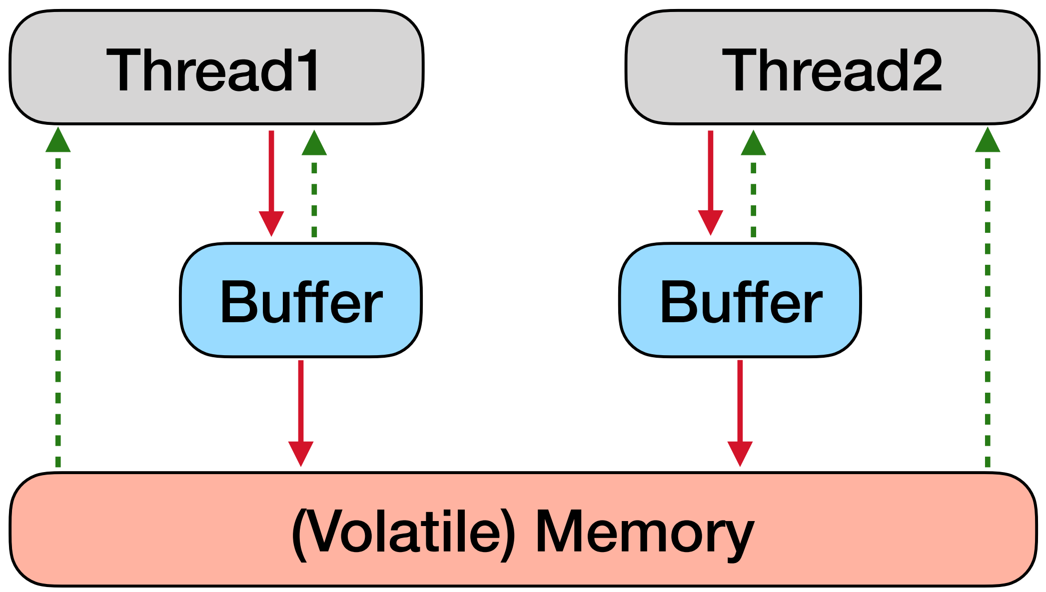
Persistent TSO (PTSO)

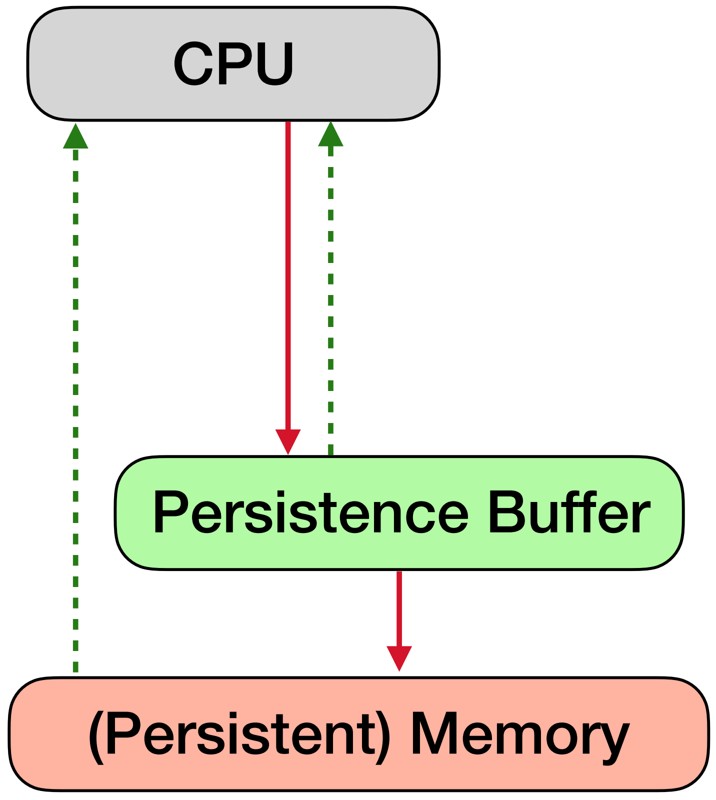




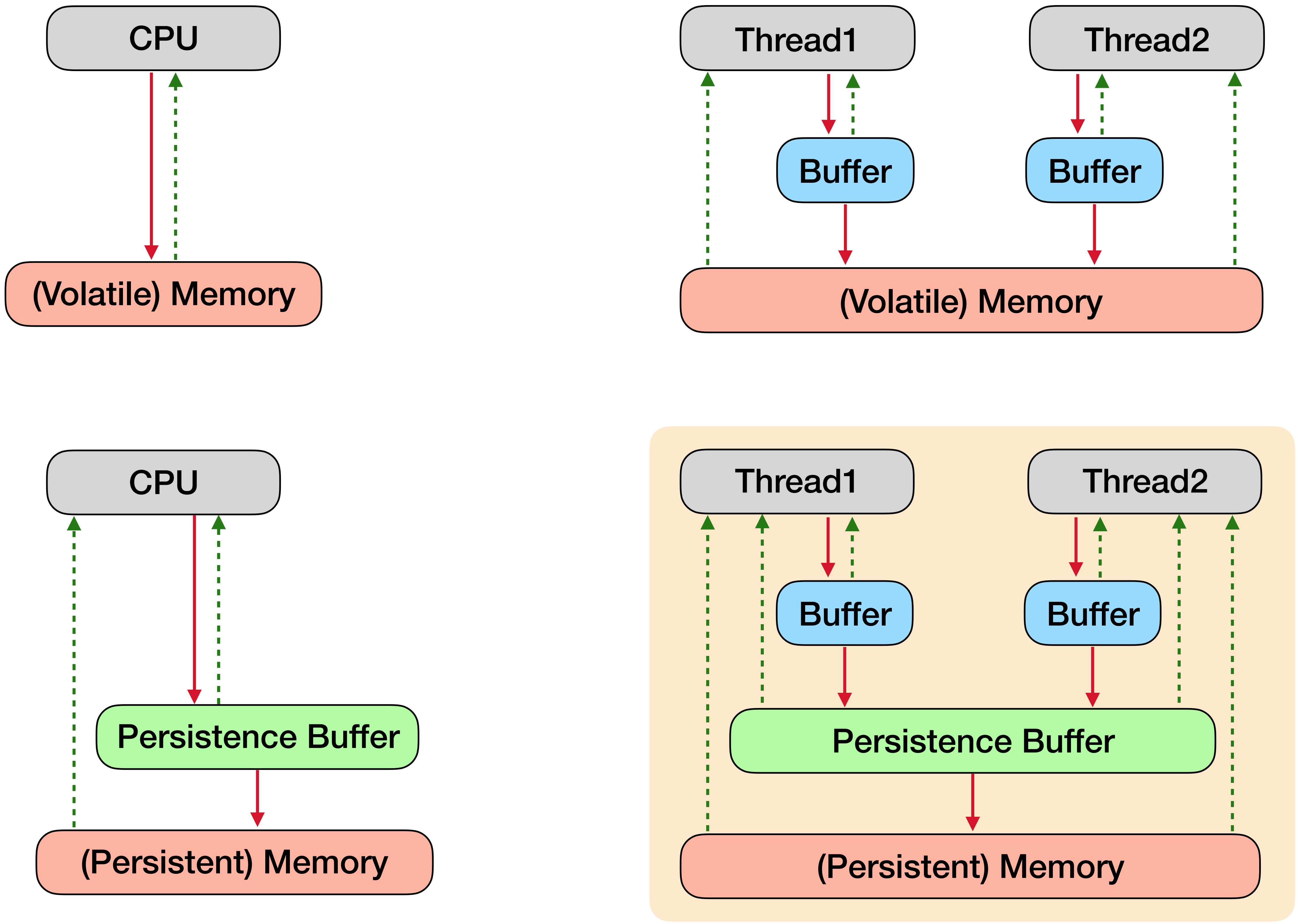
Persistent TSO (PTSO)







Persistent TSO (PTSO)



Contributions

- PTSO: First formal epoch persistency semantics under mainstream hardware
    - ***Operational*** model
    - ***Declarative*** model
    - ***Equivalence*** of the two models

Contributions

- PTSO: <u>First</u> formal epoch persistency semantics under mainstream hardware
  - ***Operational*** model
  - ***Declarative*** model
  - ***Equivalence*** of the two models
- Verifying programs under PTSO
  - PTSO programming ***pattern***
  - Correctness condition: ***persistent linearisability***
  - Verified several ***examples*** under PTSO

Contributions

- PTSO: First formal epoch persistency semantics under mainstream hardware
  - ***Operational*** model
  - ***Declarative*** model
  - ***Equivalence*** of the two models

- Verifying programs under PTSO
  - PTSO programming ***pattern***
  - Correctness condition: ***persistent linearisability***
  - Verified several ***examples*** under PTSO

Verifying programs under PTSO

```
q.enq(v) ≜
 pc:=getPC(); t:=getTC();
 n:=newNode(v,t,pc);
 map[t][pc]:=n; pfence;
 lock(q); h:=q.head;
 while (q.data[h] != null)
  h:=h+1;
 q.data[h]:=n;
 pfence; unlock(q);

q.deq() ≜
 pc:=getPC(); t:=getTC();
 lock(q);h:=q.head;n:=q.data[h];
 map[t][pc]:=n;
 if (n != null) {
  t':=n.t; pc':=n.pc;
  map[t'][pc']:=⊤ }
 pfence;
 if (n != null) {
  q.head:=h+1; pfence; }
 unlock(q); return n;

lock(q) ≜
 while (!CAS(q.lock,0,1)) skip;

unlock(q) ≜ q.lock:=0;

isIn(q,n) ≜
 h:=q.head; c:=q.data[h];
 while (c != null) {
  if (n==c) return true;
  else { h:=h+1; c:=q.data[h]; }
 } return false;

getProgress(t) ≜
 pc:=-1; n:=⊥;
 while (map[t][pc+1] !=⊥) {
  pc++; n:=map[t][pc]; }
 return (pc,n);

start() ≜
 lq:=newQueue();
 s:=P.size; lmap:=newMap(s);
 for (t in P)
  lmap[t]:=newArray(P[t].size,⊥);
 pfence;
 q:=lq; map:=lmap; run(P);

recover() ≜
 if (q==null || map==null)
  goto start();
 for(t in P) enq[t]:=-1;
 unlock(q);
 for(t in P) { // deq recovery
  (pc,n):=getProgress(t);
  if (pc>=0 && isDeq(P[t][pc])) {
   if (n==null)
    P'[t]:=sub(P[t],pc+1);
   else {
    if (inIn(q,n))
     P'[t]:=sub(P[t],pc);
    else
     P'[t]:=sub(P[t],pc+1);
    t':=n.t; pc':=n.pc;
    enq[t']:=max(enq[t'],pc'+1);}
  }
  else if (pc<0) P'[t]:=P[t];
 }
 for(t in P) { // enq recovery
  (pc,n):=getProgress(t);
  if (pc>=0 && isEnq(P[t][pc])) {
   if (pc < enq[t])
    P'[t]:=sub(P[t],enq[t]);
   else if (n==⊤ || isIn(q,n))
    P'[t]:=sub(P[t],pc+1);
   else
    P'[t]:=sub(P[t],pc); }
 } run(P');
```

The ***persistent*** variant of the Michael-Scott queue and its ***recovery*** mechanism

Verifying programs under PTSO

```
q.enq(v) ≜
 pc:=getPC(); t:=getTC();
 n:=newNode(v,t,pc);
 map[t][pc]:=n; pfence;
 lock(q); h:=q.head;
 while (q.data[h] != null)
  h:=h+1;
 q.data[h]:=n;
 pfence; unlock(q);

q.deq() ≜
 pc:=getPC(); t:=getTC();
 lock(q);h:=q.head;n:=q.data[h];
 map[t][pc]:=n;
```

```
unlock(q) ≜ q.lock:=0;

isIn(q,n) ≜
 h:=q.head; c:=q.data[h];
 while (c != null) {
  if (n==c) return true;
  else { h:=h+1; c:=q.data[h]; }
 } return false;

getProgress(t) ≜
 pc:=-1; n:=⊥;
 while (map[t][pc+1] != ⊥){
  pc++; n:=map[t][pc]; }
 return (pc,n);
```

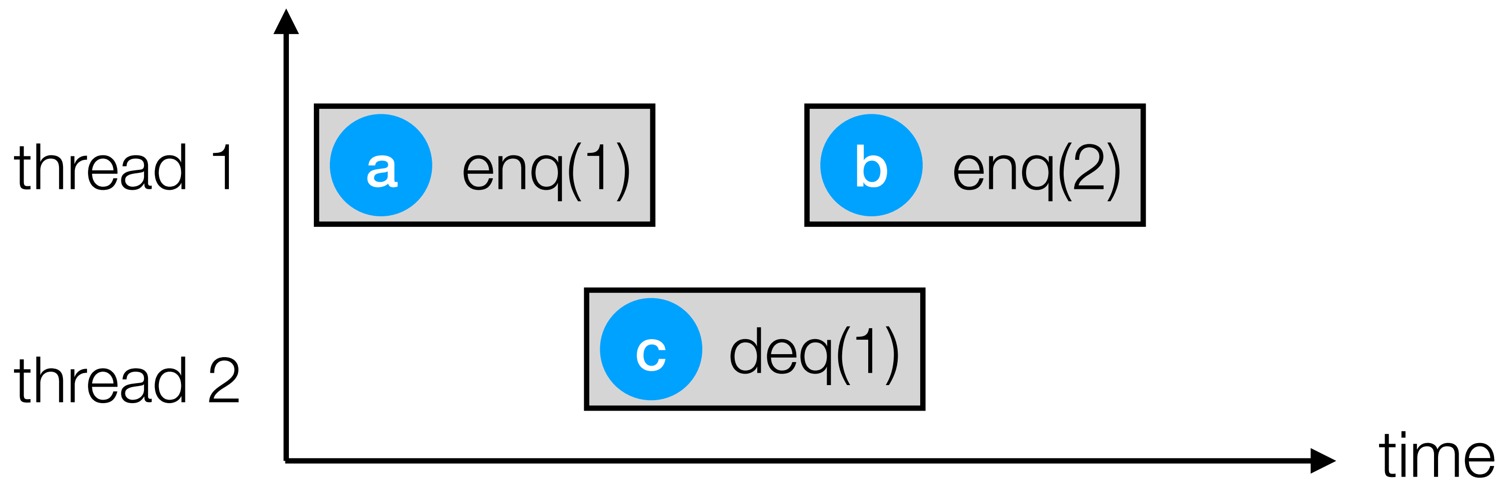
```
start() ≜
 lq:=newQueue();
 s:=P.size; lmap:=newMap(s);
 for (t in P)
  lmap[t]:=newArray(P[t].size,⊥);
 pfence;
 q:=lq; map:=lmap; run(P);

recover() ≜
 if (q==null || map==null)
  goto start();
 for(t in P) enq[t]:=-1;
 unlock(q);
 for(t in P) { // deq recovery
```

What constitutes a *correct persistent* implementation?

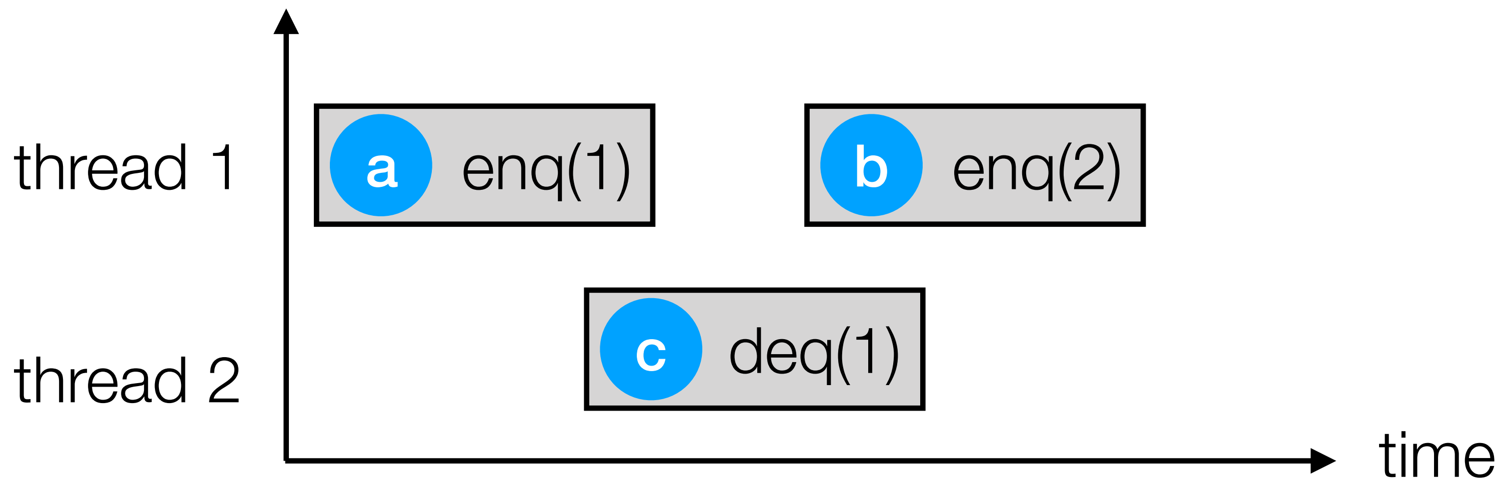
```
     enq[t']:=max(enq[t'],pc'+1);}
  }
  else if (pc<0) P'[t]:=P[t];
 }
 for(t in P) { // enq recovery
  (pc,n):=getProgress(t);
  if (pc>=0 && isEnq(P[t][pc])){
   if (pc < enq[t])
    P'[t]:=sub(P[t],enq[t]);
   else if (n==⊤ || isIn(q,n))
    P'[t]:=sub(P[t],pc+1);
   else
    P'[t]:=sub(P[t],pc); }
 } run(P');
```

The ***persistent*** variant of the Michael-Scott queue and its ***recovery*** mechanism

Linearisability

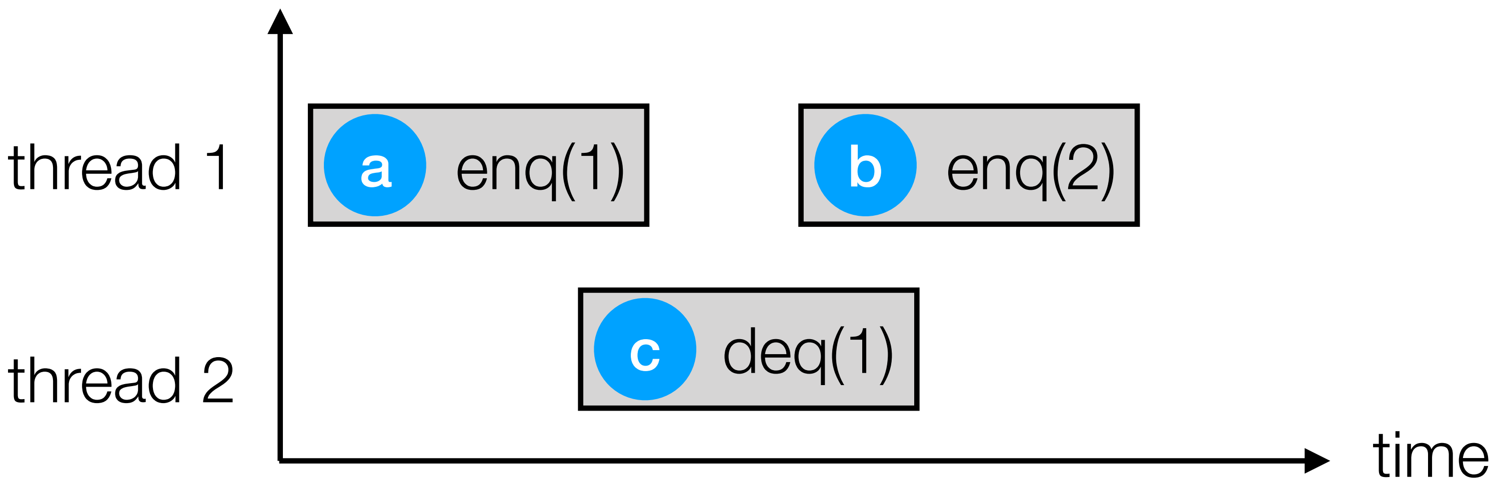


Linearisability



- Define happens-before relation *hb*
  - $(e_1, e_2) \in$ *hb* $\iff$ e_1.end $<_{\text{time}}$ e_2.begin

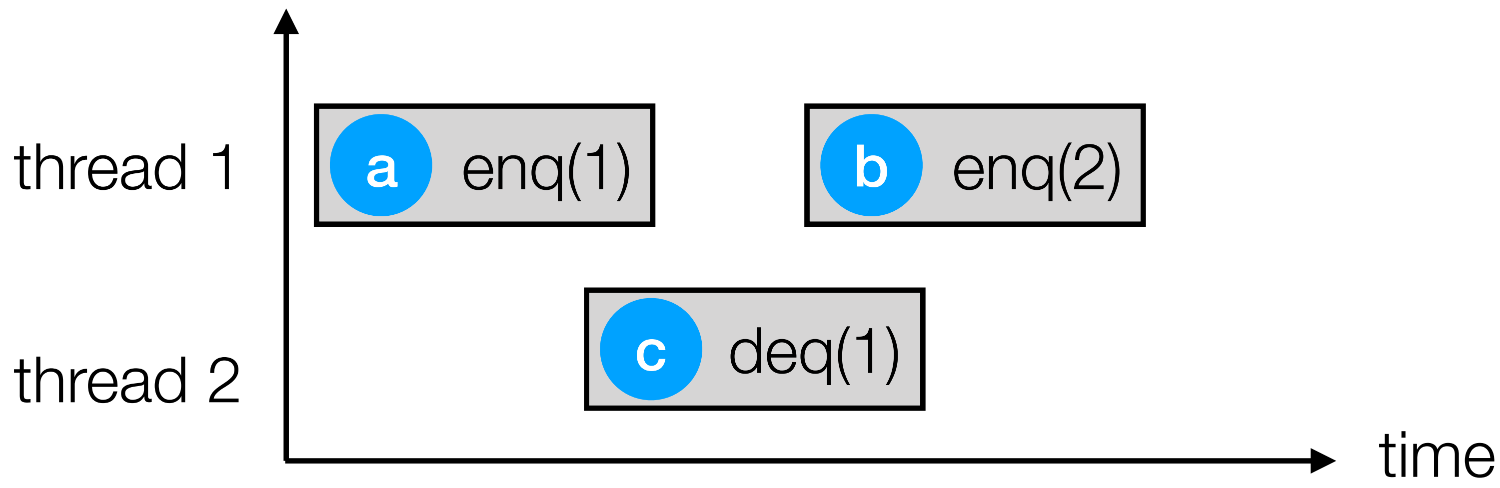
Linearisability



- Define happens-before relation *hb*
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin}$

    -- e.g. (a, b) ∈ *hb*   (a, c) ∉ *hb*

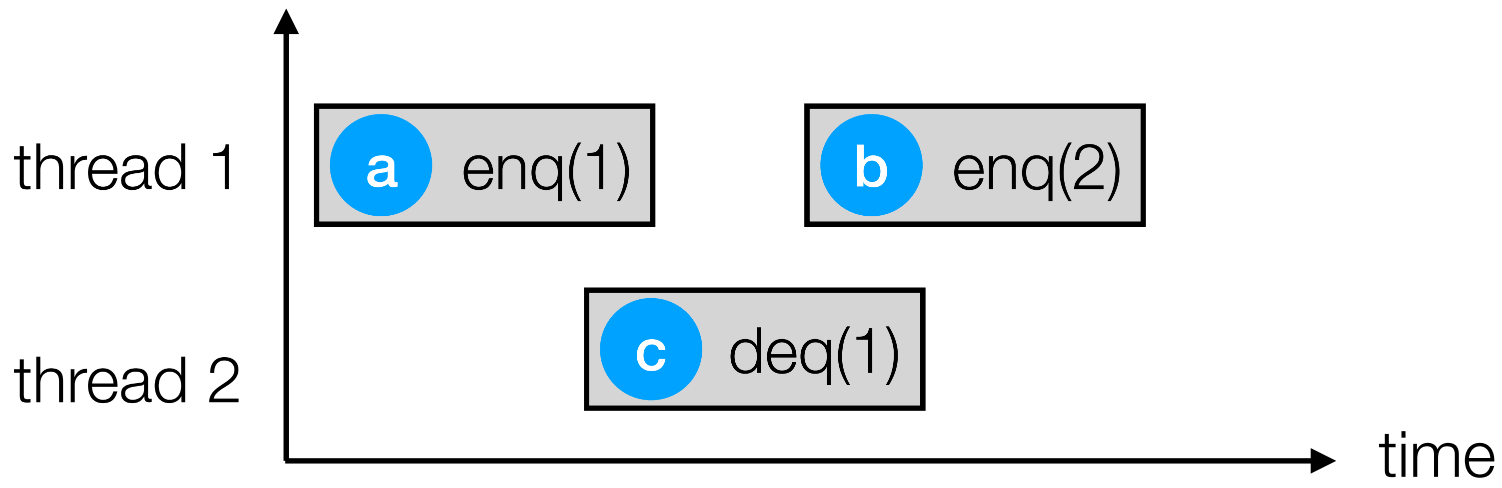
Linearisability



- Define happens-before relation *hb*
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\textbf{time}} e_2.\text{begin}$

    -- e.g. (a, b) $\in$ *hb*   (a, c) $\notin$ *hb*
- ***Linearisable*** $\iff \exists$ *H*. *H* totally orders events
  - *H* respects *hb*
  - *H* is a ***legal*** sequence (library-specific)

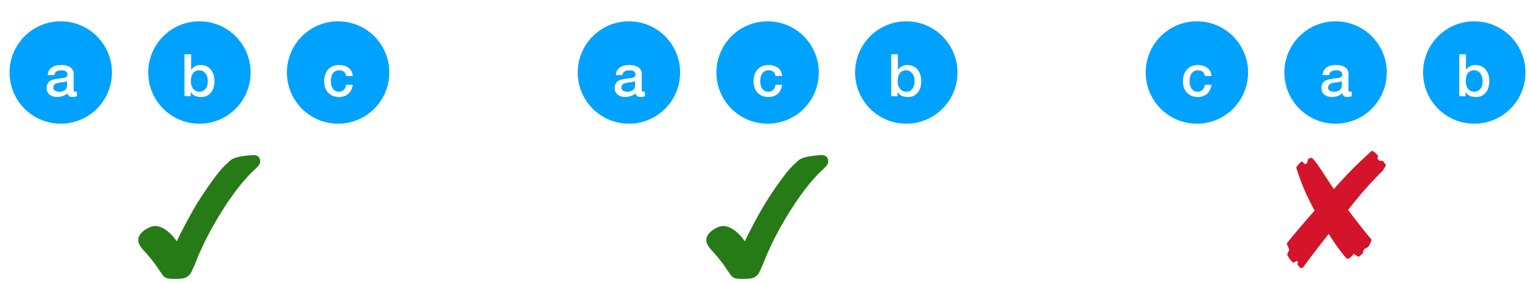
Linearisability



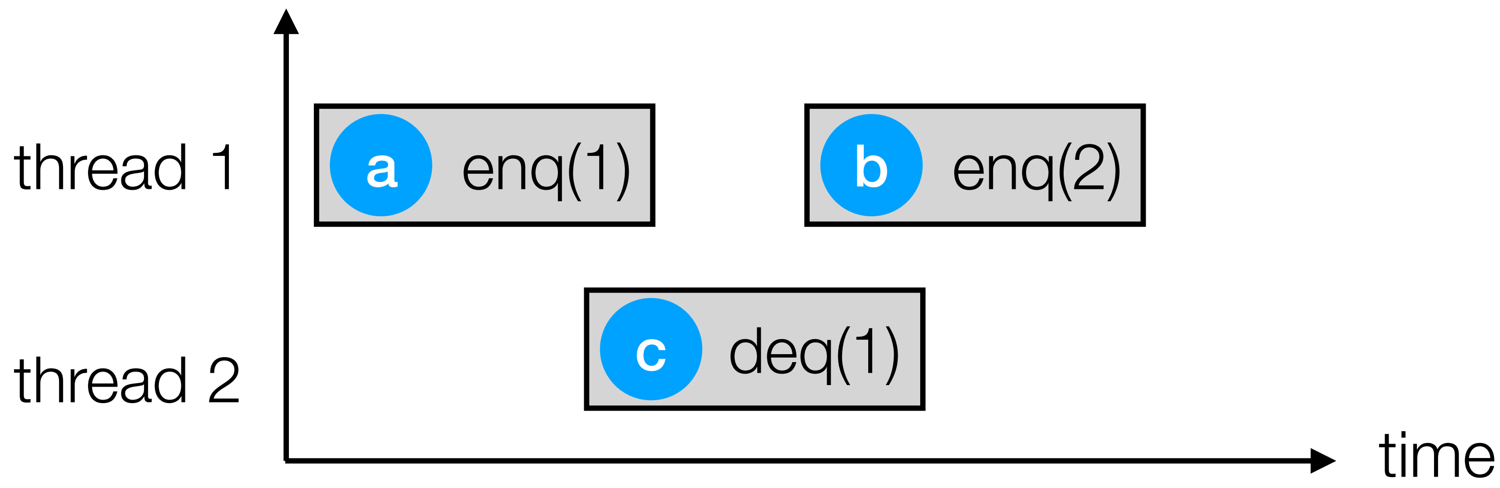
- Define happens-before relation *hb*
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\mathbf{time}} e_2.\text{begin}$

    -- e.g. (a, b) $\in$ *hb* (a, c) $\notin$ *hb*
- ***Linearisable*** $\iff \exists$ *H*. *H* totally orders events
  - *H* respects *hb*
  - *H* is a ***legal*** sequence (library-specific)

    -- e.g. FIFO sequences for queue



Linearisability



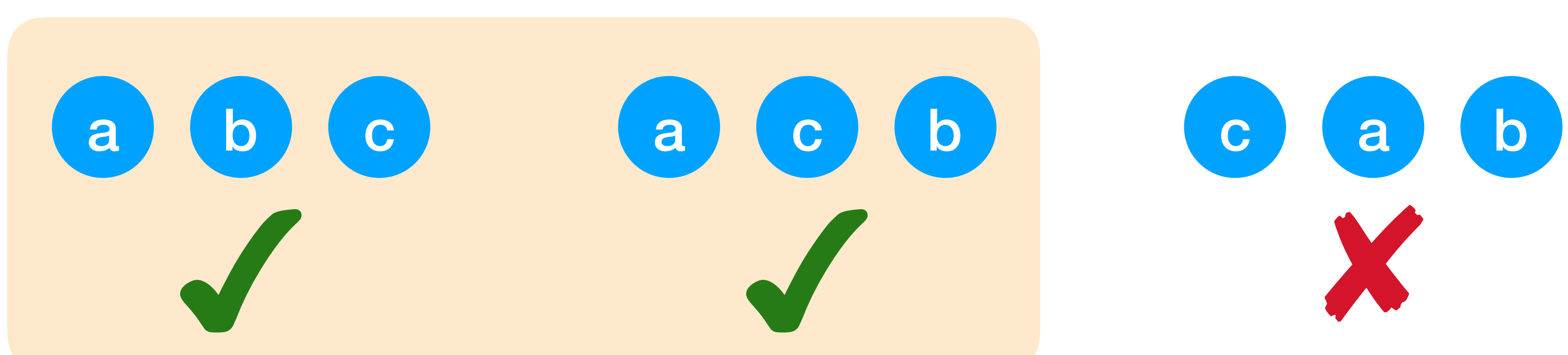
- Define happens-before relation *hb*
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin}$

    -- e.g. (a, b) $\in$ *hb* (a, c) $\notin$ *hb*

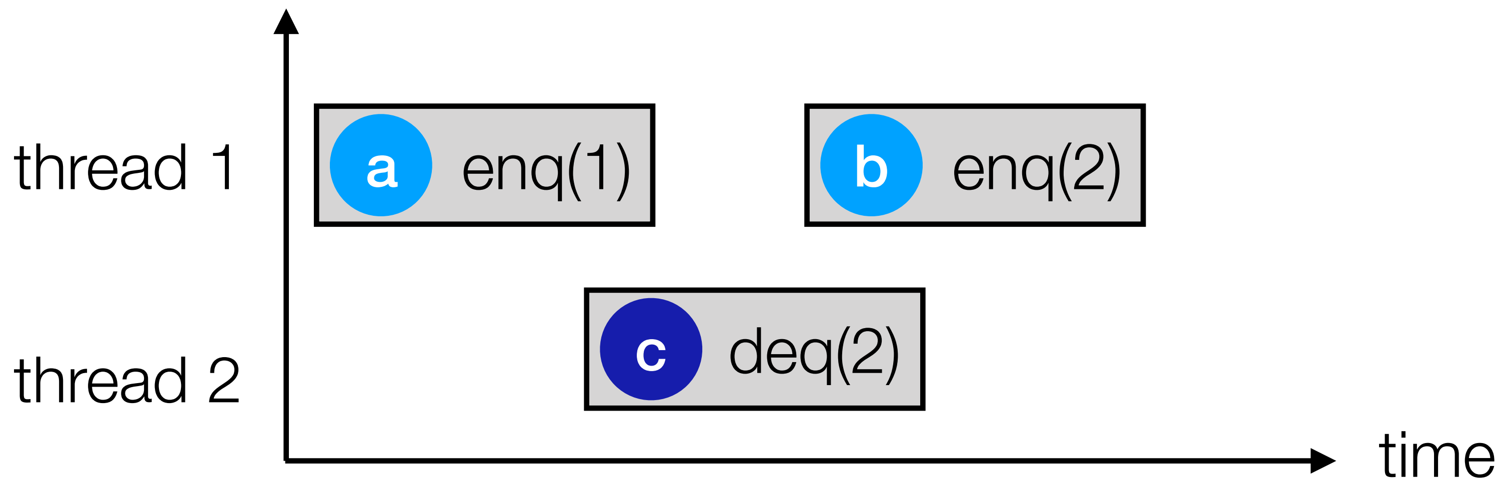
- ***Linearisable*** $\iff \exists$ *H*. *H* totally orders events
  - *H* respects *hb*
  - *H* is a ***legal*** sequence (library-specific)

    -- e.g. FIFO sequences for queue

linearisable



Linearisability

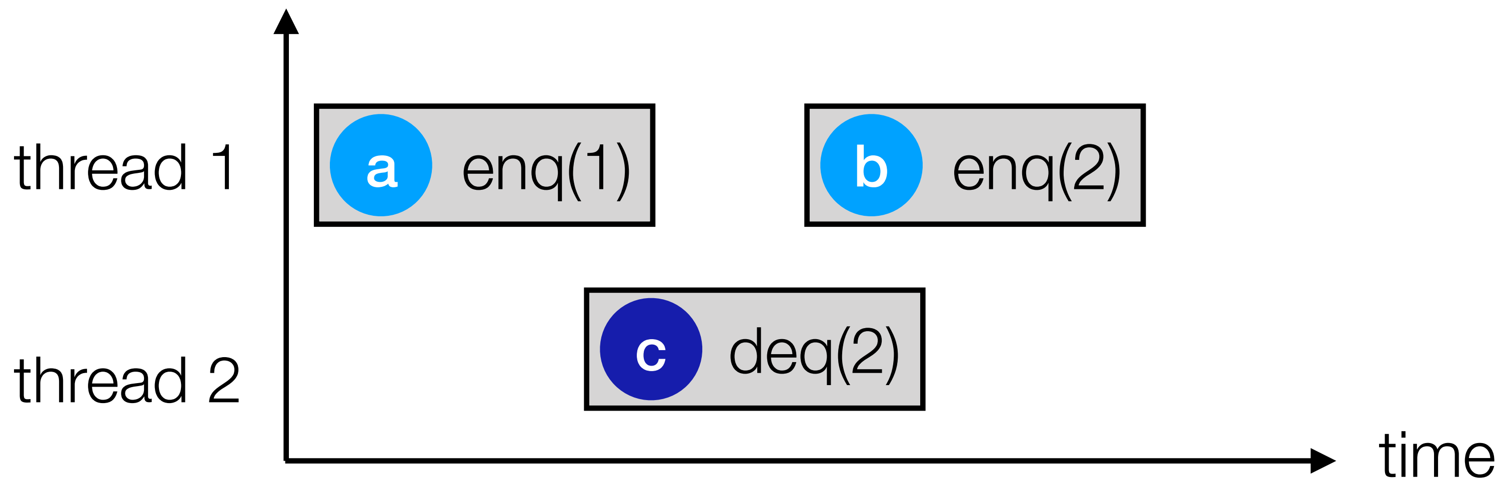


- Define happens-before relation *hb*
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin}$

    -- e.g. (a, b) $\in$ *hb* (a, c) $\notin$ *hb*
- ***Linearisable*** $\iff \exists H.$ *H* totally orders events
  - *H* respects *hb*
  - *H* is a ***legal*** sequence (library-specific)

    -- e.g. FIFO sequences for queue

Linearisability



- Define happens-before relation *hb*
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin}$

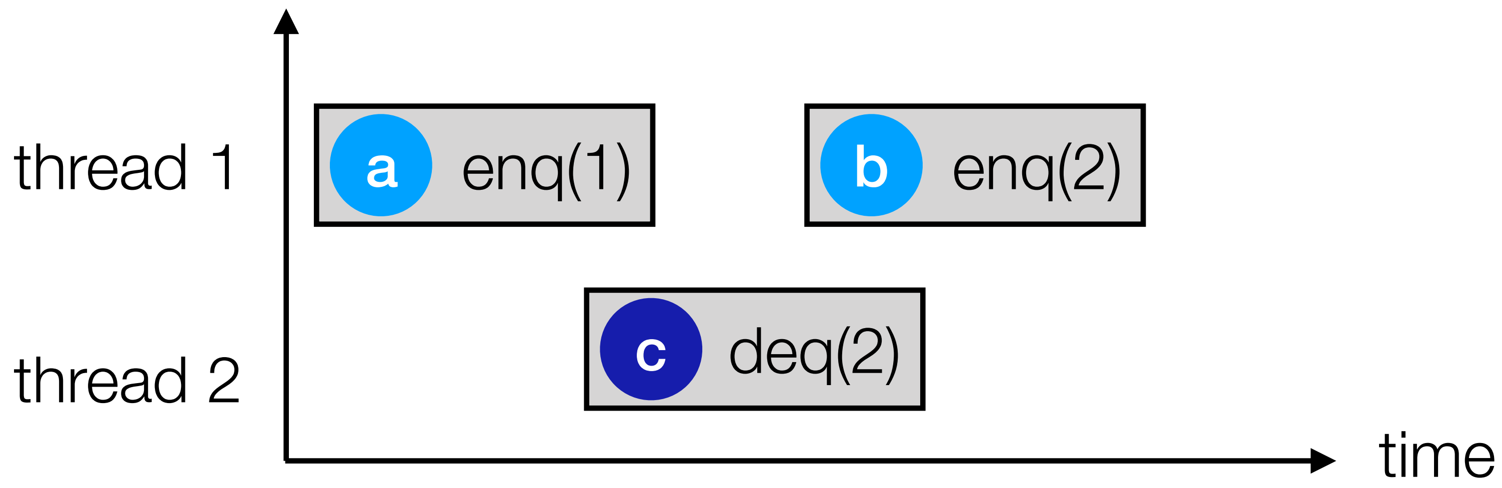
    -- e.g. (a, b) $\in$ *hb* (a, c) $\notin$ *hb*

- ***Linearisable*** $\iff \exists$ *H*. *H* totally orders events
  - *H* respects *hb*
  - *H* is a ***legal*** sequence (library-specific)

    -- e.g. FIFO sequences for queue

a b c   a c b   c a b

Linearisability



- Define happens-before relation *hb*
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\textbf{time}} e_2.\text{begin}$

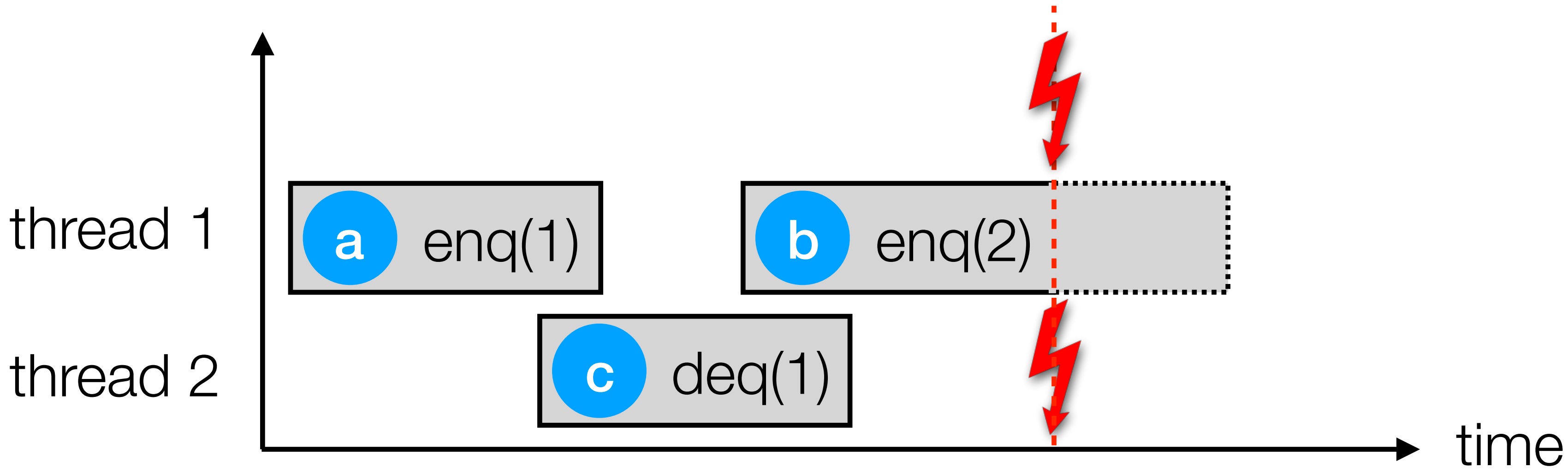
    -- e.g. (a, b) ∈ *hb*   (a, c) ∉ *hb*
- ***Linearisable*** ⟺ ∃ *H*. *H* totally orders events
  - *H* respects *hb*
  - *H* is a ***legal*** sequence (library-specific)

    -- e.g. FIFO sequences for queue

non-linearisable
(not legal)

a b c ✗   a c b ✗   c a b ✗

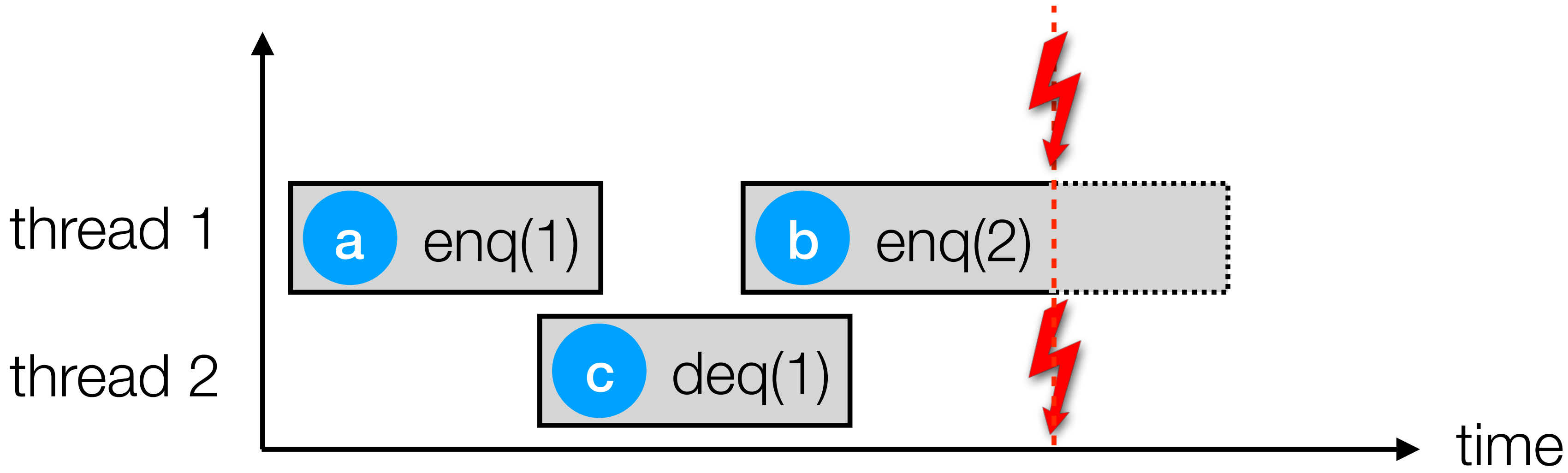
Persistent Linearisability



- Define happens-before relation *hb*
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin}$

    -- e.g. (a , b) ∈ *hb* (a , c) ∉ *hb*

Persistent Linearisability

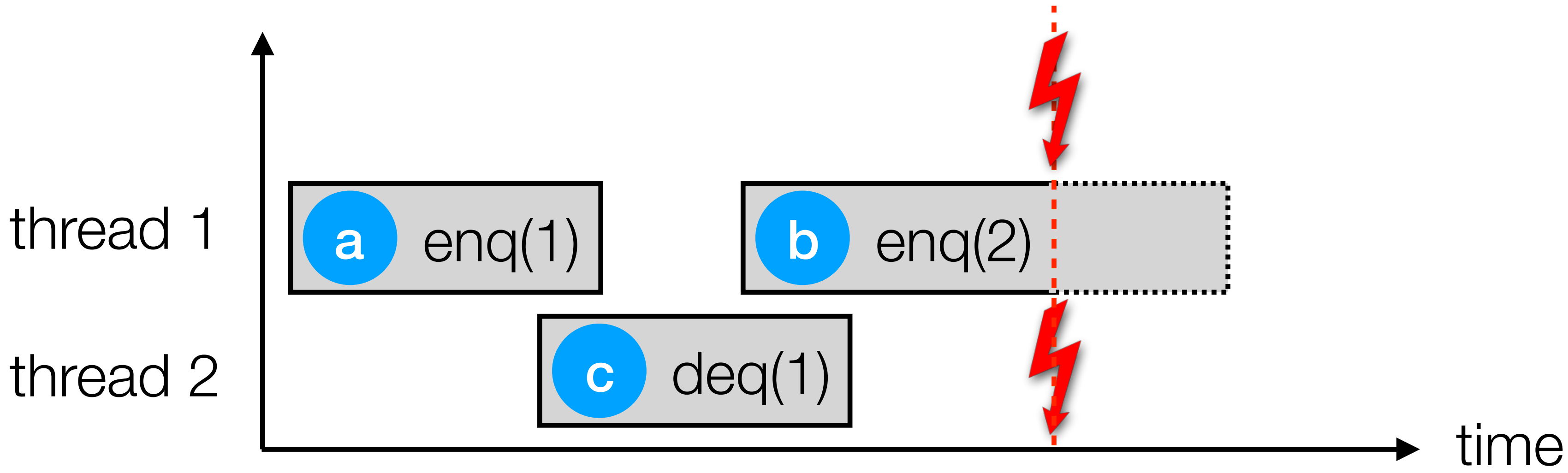


- Define happens-before relation *hb*
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin}$

    -- e.g. (a, b) ∈ *hb* (a, c) ∉ *hb*

- ***Persistently linearisable*** ⟺ ∃ *H*. *H* totally orders a ***subset*** *S* of events
  - *H* respects *hb*
  - *H* is a legal sequence

Persistent Linearisability



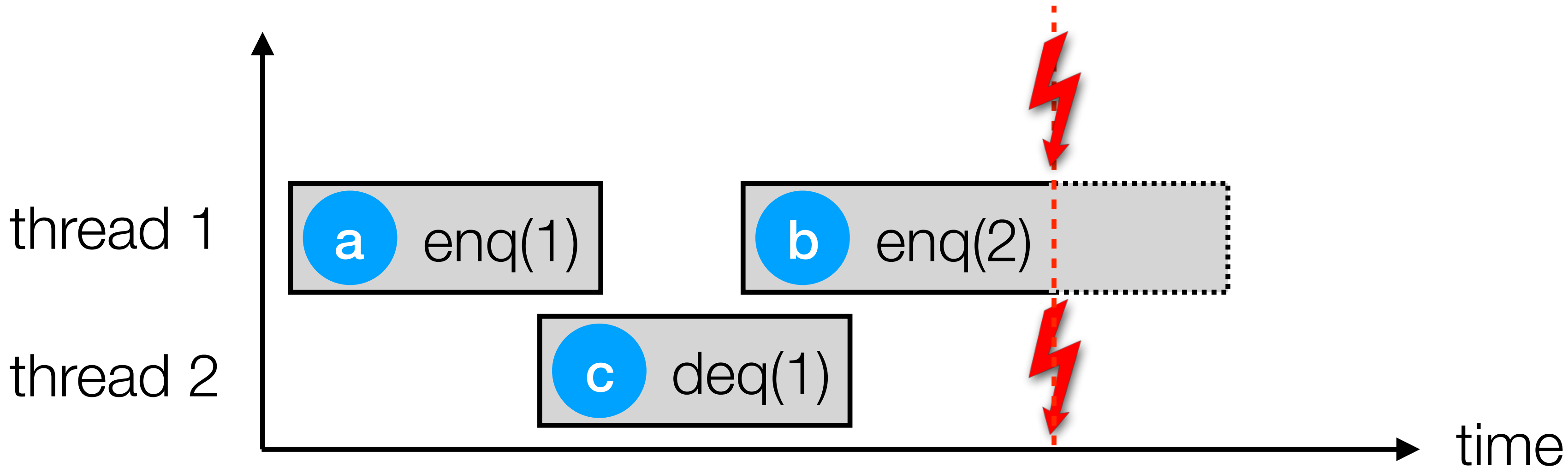
- Define happens-before relation *hb*
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\textbf{time}} e_2.\text{begin}$

    -- e.g. (a, b) ∈ *hb* (a, c) ∉ *hb*

- ***Persistently linearisable*** ⟺ ∃ *H*. *H* totally orders a ***subset*** *S* of events
  - *H* respects *hb*
  - *H* is a legal sequence
  - *S* is *hb-**prefix-closed*** : $(a, b) \in hb$ and $b \in S \Rightarrow a \in S$

    -- persists are ***asynchronous***: only a ***prefix*** may persist after a crash

Persistent Linearisability

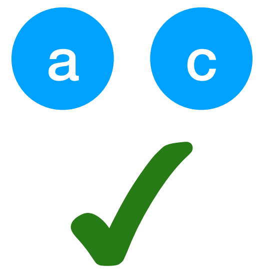


- Define happens-before relation *hb*
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin}$

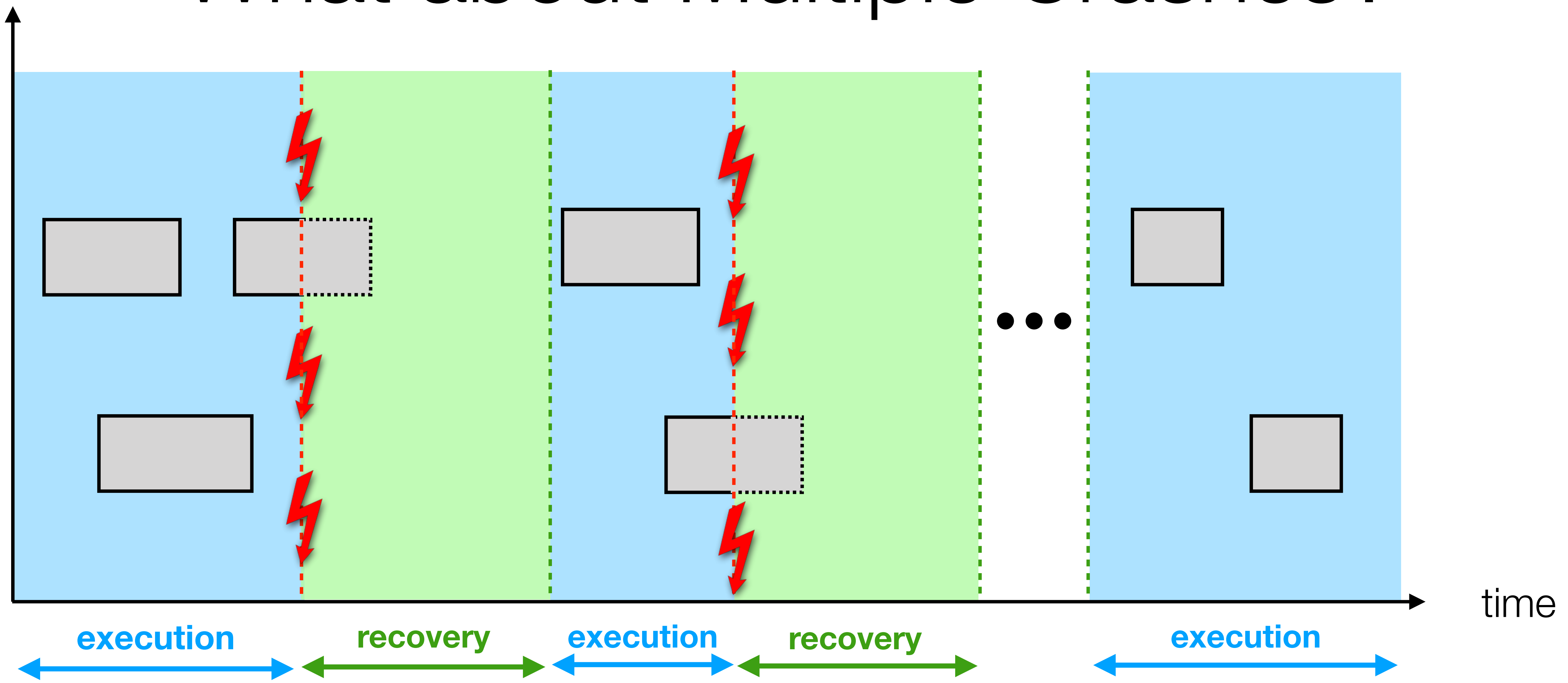
    -- e.g. (a, b) ∈ *hb*   (a, c) ∉ *hb*
- ***Persistently linearisable*** $\iff \exists$ *H*. *H* totally orders a ***subset*** *S* of events
  - *H* respects *hb*
  - *H* is a legal sequence
  - *S* is *hb-**prefix-closed*** : $(a, b) \in hb$ and $b \in S \Rightarrow a \in S$

    -- persists are ***asynchronous***: only a ***prefix*** may persist after a crash

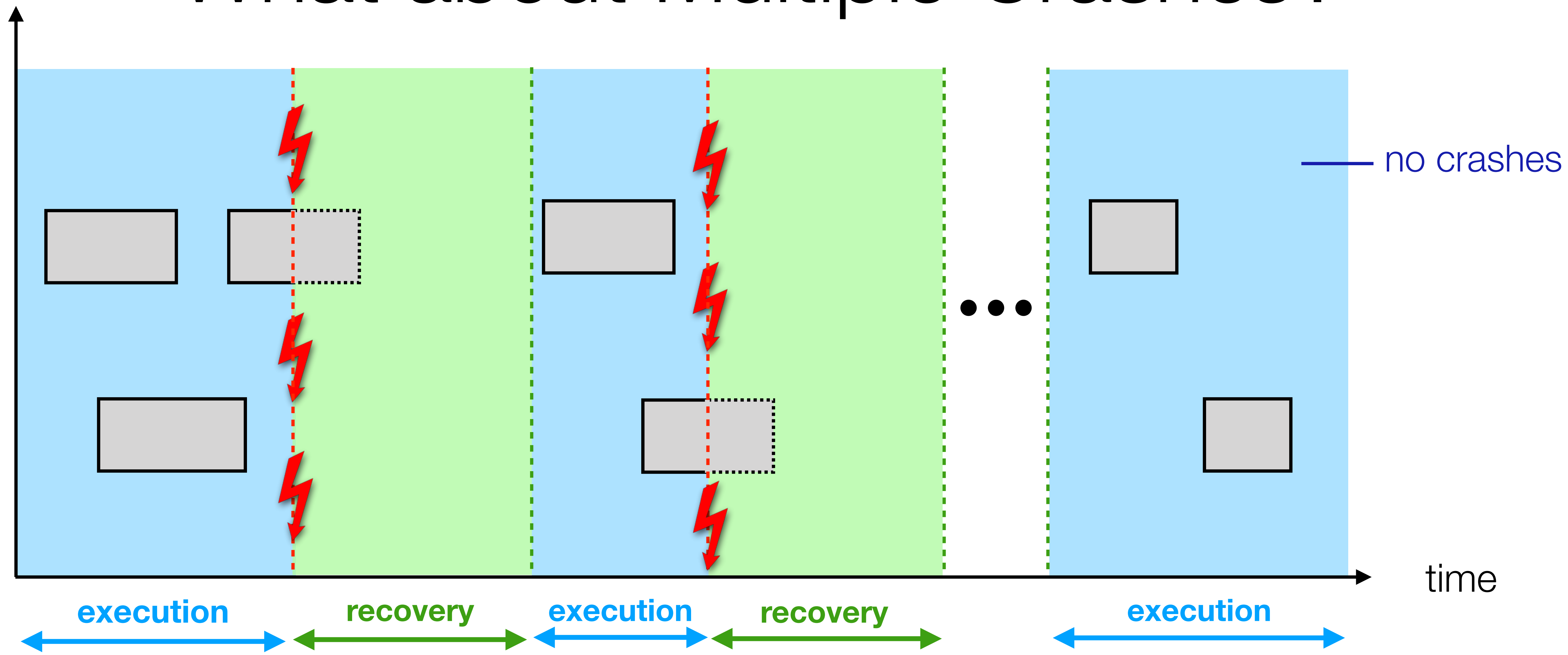
Persistently linearisable



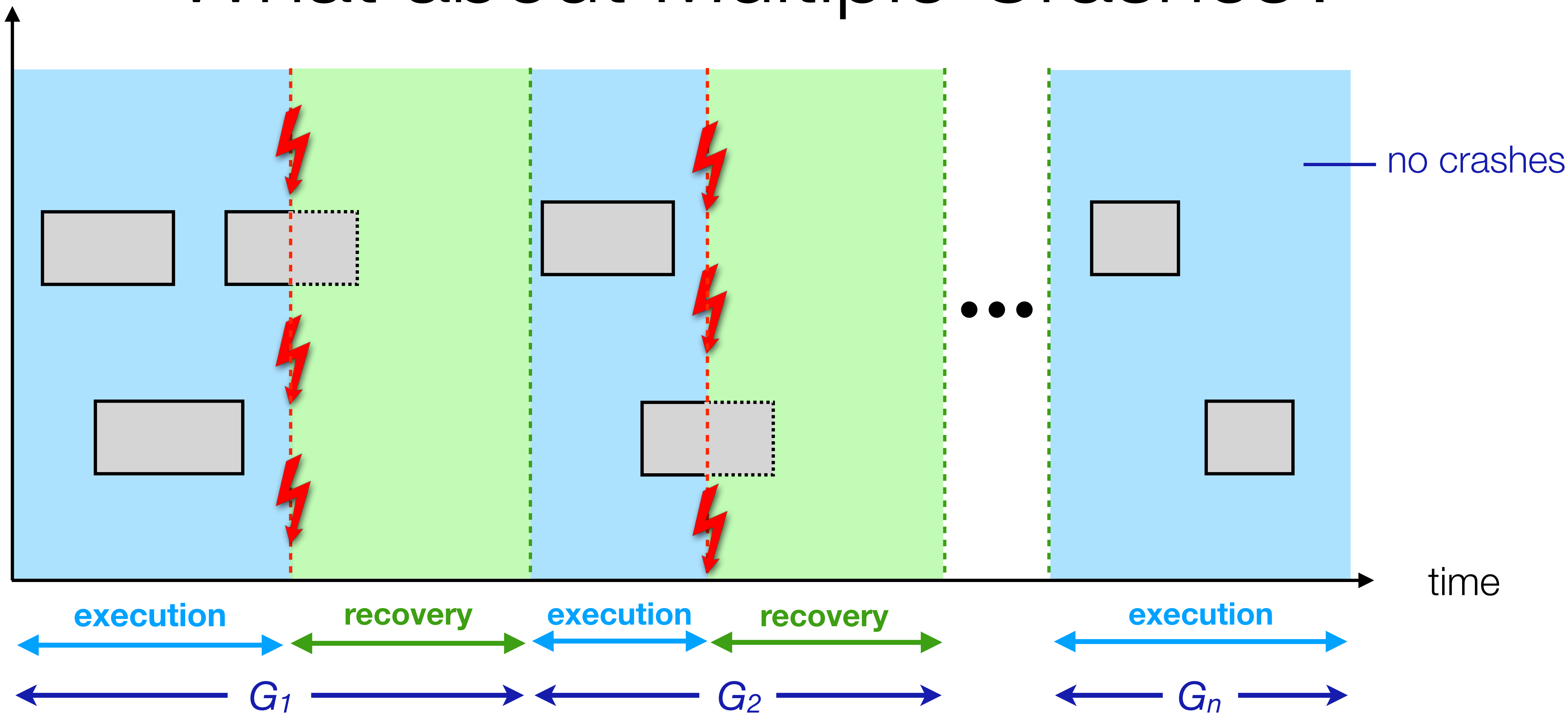
What about Multiple Crashes?



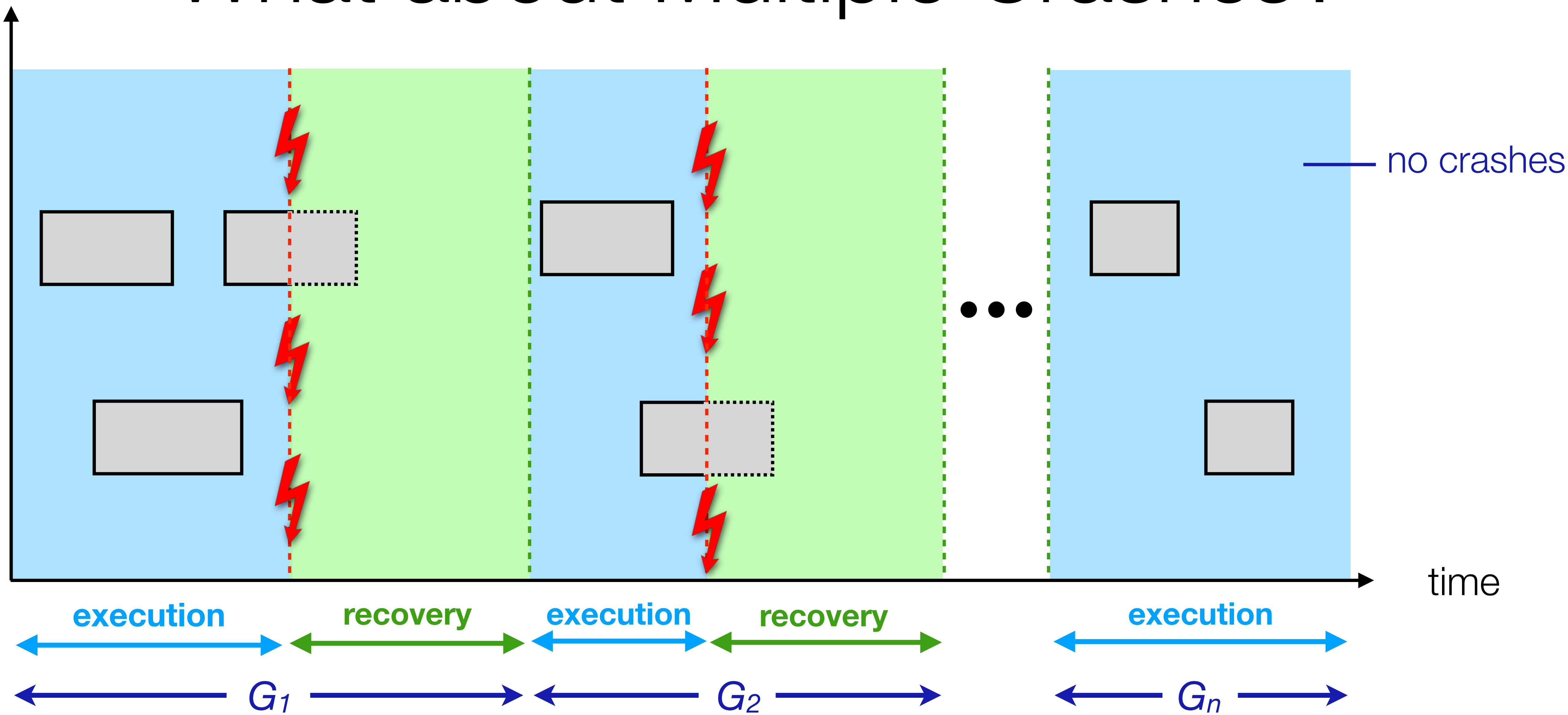
What about Multiple Crashes?



What about Multiple Crashes?



What about Multiple Crashes?



- A ***chain*** $G_1 \cdots G_n$ is ***persistently linearisable*** $\iff \exists\, H_1 \cdots H_n$.
  - H_i persistently linearises G_i — as before
  - H_1 ++ ... ++ H_n is a legal sequence

Conclusions

- PTSO: <u>First</u> formal epoch persistency semantics under mainstream hardware
  - ***Operational*** model
  - ***Declarative*** model
  - ***Equivalence*** of the two models
- Verifying programs under PTSO
  - PTSO programming ***pattern***
  - Correctness condition: ***persistent linearisability***
  - Verified several ***examples*** under PTSO

Conclusions

- PTSO: First formal epoch persistency semantics under mainstream hardware
  - ***Operational*** model
  - ***Declarative*** model
  - ***Equivalence*** of the two models
- Verifying programs under PTSO
  - PTSO programming ***pattern***
  - Correctness condition: ***persistent linearisability***
  - Verified several ***examples*** under PTSO

Thank you for listening!

azalea@mpi-sws.org SoundAndComplete.org @azalearaad

Programming Pattern

```
1. // log progress
2. pfence
3. // do the work
4. pfence
```

Programming Pattern

```
1. // log progress
2. pfence
3. // do the work
4. pfence
```

Log ***at most one step*** ahead of work

Programming Pattern

```
1. // log progress
2. pfence
3. // do the work
4. pfence
```

Log ***at most one step*** ahead of work

```
     q.enq(v) ≜
1.     pc:=getPC(); t:=getTC();
       n:=newNode(v,t,pc,null);
       map[t][pc]:=n;
2.     pfence;
3.     h:=q.head;
       find: while (q.data[h] != null)
         h:=h+1;
       if (!CAS(q.data[h],null,n))
         goto find;
4.     pfence;
```