Lessons that I learned about linearizability

Hongseok Yang (University of Oxford)
With many colleagues.
In particular, Alexey Gotsman (IMDEA, Spain)

Linearizability

- Widely used correctness condition for concurrent libraries (i.e., data structures).
- Usually expresses a relationship between concurrent library and sequential library.

Personal motivation

- Learned the definition of linearizability from Attiya's talk at Cambridge in 2008.
- Mystified.
- Motivated me to study linearizability using tools from programming languages.

Tools from PL

- PL researchers like to define concepts from end users' perspective (aka observations).
- A concurrent object L_0 observationally refines L_1 iff for every client program C, $Obs(C[L_0]) \subseteq Obs(C[L_1])$.

Lessons that I learned

- Defining linearizability for a new problem amounts to defining histories and happen-before order.
- The definition of histories captures how clients communicate with concurrent libraries.
- Connection between happen-before and dependency.
- Sanity checks.
  - Gotsman's composition/decomposition --- histories.
  - Rearrangement lemma --- happen-before order.
- Importance of well-formed definable traces.

Lessons that I learned

- Defining linearizability for a new problem amounts to defining histories and happen-before order.
- The definition of histories captures how clients communicate with concurrent libraries.
- Connection between happen-before and dependency.
- Sanity checks.
  - Gotsman's composition/decomposition --- histories.
  - Rearrangement lemma --- happen-before order.
- Importance of well-formed definable traces.

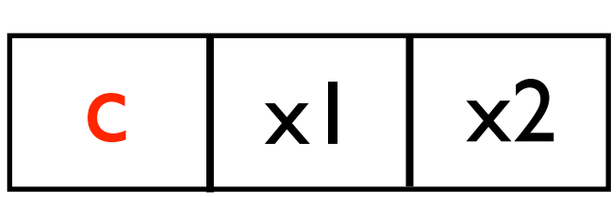
Expected outcome

- Understand some aspect of linearizability from PL-researchers' perspective.
- In particular, the connection between happen-before and dependency.
- Be able to propose an appropriate modification of linearizability, when attacking a new problem.

Review of linearizability

Seqlock

- Atomic reads and writes to two memory locations.



- Every new value will have a new version number c.

Implementation of Seqlock

```
init() { c = x1 = x2 = 0; }

write(in word d1, in word d2) {
  c++;
  x1 = d1; x2 = d2;
  c++;
}

read(out word d1, out word d2) {
  word c0;
  do {
    do { c0 = c; } while (c0%2);
    d1 = x1; d2 = x2;
  } while (c != c0);
}
```

Implementation of Seqlock

```
init() { c = x1 = x2 = 0; }

write(in word d1, in word d2) {
  c++;
  x1 = d1; x2 = d2;
  c++;
}

read(out word d1, out word d2) {
  word c0;
  do {
    do { c0 = c; } while (c0%2);
    d1 = x1; d2 = x2;
  } while (c != c0);
}
```

Odd c: writing in progress

Implementation of Seqlock

```
init() { c = x1 = x2 = 0; }

write(in word d1, in word d2) {
  c++;
  x1 = d1; x2 = d2;
  c++;
}

read(out word d1, out word d2) {
  word c0;
  do {
    do { c0 = c; } while (c0%2);
    d1 = x1; d2 = x2;
  } while (c != c0);
}
```

Odd c: writing in progress

Even c: valid values

Implementation of Seqlock

```
init() { c = x1 = x2 = 0; }

write(in word d1, in word d2) {
  c++;
  x1 = d1; x2 = d2;
  c++;
}

read(out word d1, out word d2) {
  word c0;
  do {
    do { c0 = c; } while (c0%2);
    d1 = x1; d2 = x2;
  } while (c != c0);
}
```

Odd c: writing in progress

Even c: valid values

Wait until c is even: no writing in progress

Implementation of Seqlock

```
init() { c = x1 = x2 = 0; }

write(in word d1, in word d2) {
  c++;
  x1 = d1; x2 = d2;
  c++;
}

read(out word d1, out word d2) {
  word c0;
  do {
    do { c0 = c; } while (c0%2);
    d1 = x1; d2 = x2;
  } while (c != c0);
}
```

Odd c: writing in progress

Even c: valid values

Wait until c is even: no writing in progress

Read the data

Implementation of Seqlock

```
init() { c = x1 = x2 = 0; }

write(in word d1, in word d2) {
  c++;
  x1 = d1; x2 = d2;
  c++;
}

read(out word d1, out word d2) {
  word c0;
  do {
    do { c0 = c; } while (c0%2);
    d1 = x1; d2 = x2;
  } while (c != c0);
}
```

Odd c: writing in progress

Even c: valid values

Wait until c is even: no writing in progress

Read the data

Check c didn't change: got a consistent snapshot

Specification of Seqlock

```
init() { c = x1 = x2 = 0; }

write(in word d1, in word d2) {
  c++;
  x1 = d1; x2 = d2;
  c++;
}

read(out word d1, out word d2) {
  word c0;
  do {
    do { c0 = c; } while (c0%2);
    d1 = x1; d2 = x2;
  } while (c != c0);
}
```

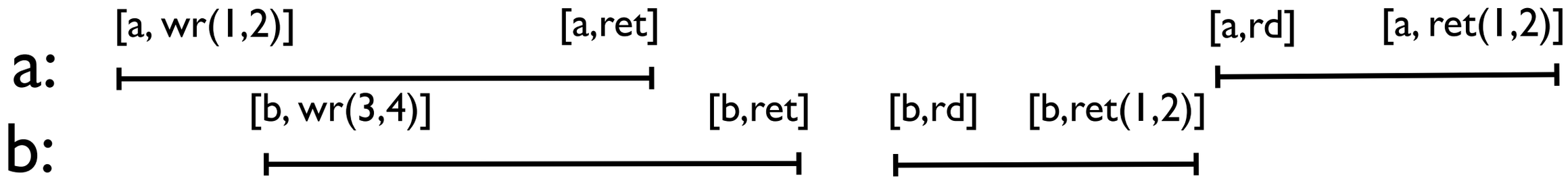
```
init() { x1 = x2 = 0; }

write(in word d1, in word d2) {
  atomic { x1 = d1; x2 = d2; }
}

read(out word d1, out word d2) {
  atomic { d1 = x1; d2 = x2; }
}
```

Linearizability of Seqlock

- Every execution of Seqlock corresponds to some execution of Spec.
- Every history h1 ∈ [Seqlock] is related to some history h2 ∈ [Spec] by LinH.

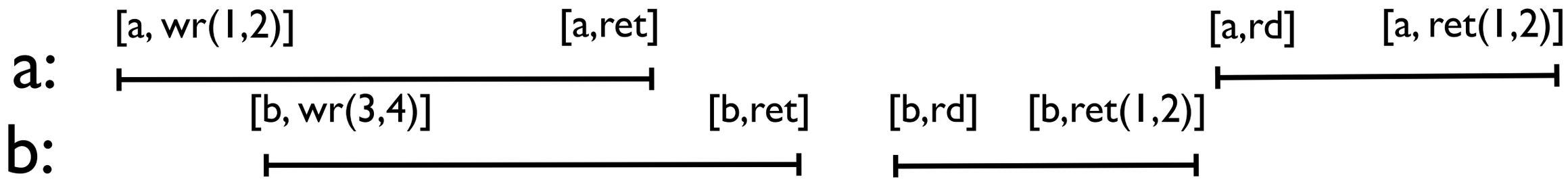
Linearizability of Seqlock

- Execution of Seqlock.



Linearizability of Seqlock

- Execution of Seqlock.

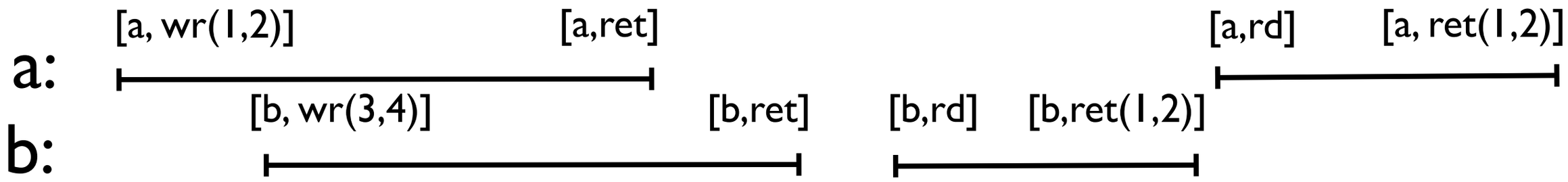


- Get the history: Keep call & return actions only.

[a,wr(1,2)][b,wr(3,4)][a,ret][b,ret][b,rd][b,ret(1,2)][a,rd][a,ret(1,2)]

Linearizability of Seqlock

- Execution of Seqlock.



- Get the history: Keep call & return actions only.

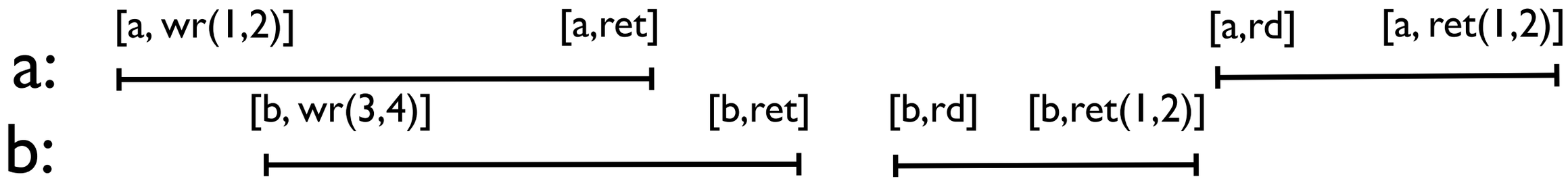
[a,wr(1,2)][b,wr(3,4)][a,ret][b,ret][b,rd][b,ret(1,2)][a,rd][a,ret(1,2)]

  - Find a LinH-related history in [Spec]: Permute actions, while keeping thread & happen-before order.

[b,wr(3,4)][b,ret][a,wr(1,2)][a,ret][b,rd][b,ret(1,2)][a,rd][a,ret(1,2)]

Linearizability of Seqlock

- Execution of Seqlock.



- Get the history: Keep call & return actions only.

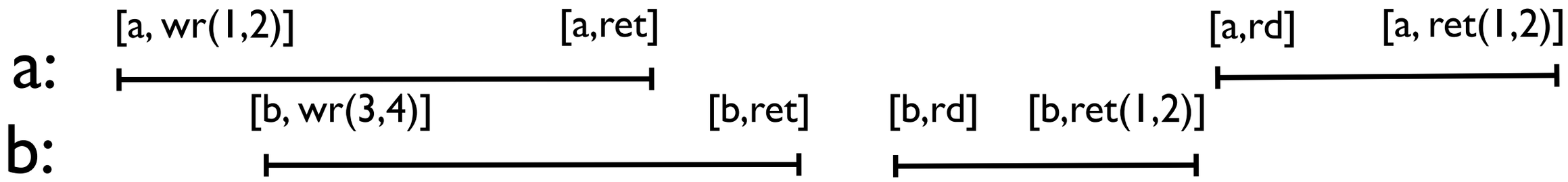
[a,wr(1,2)][b,wr(3,4)][a,ret][b,ret][b,rd][b,ret(1,2)][a,rd][a,ret(1,2)]

  - Find a LinH-related history in [Spec]: Permute actions, while keeping thread & happen-before order.

[b,wr(3,4)][b,ret][a,wr(1,2)][a,ret][b,rd][b,ret(1,2)][a,rd][a,ret(1,2)]

Linearizability of Seqlock

- Execution of Seqlock.



- Get the history: Keep call & return actions only.

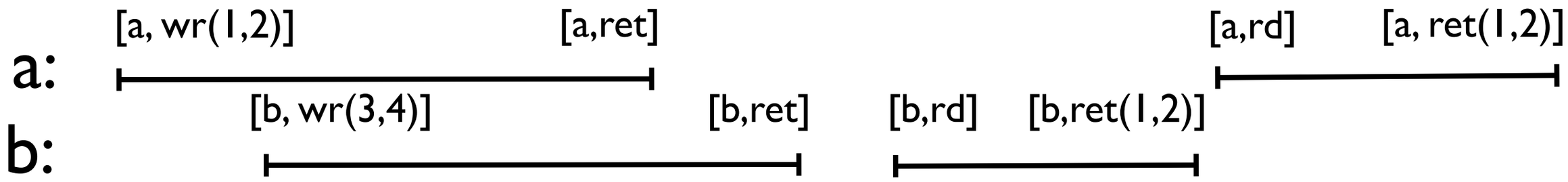
[a,wr(1,2)][b,wr(3,4)][a,ret][b,ret][b,rd][b,ret(1,2)][a,rd][a,ret(1,2)]

- Find a LinH-related history in [Spec]: Permute actions, while keeping thread & happen-before order.

[b,wr(3,4)][b,ret][a,wr(1,2)][a,ret][b,rd][b,ret(1,2)][a,rd][a,ret(1,2)]

Linearizability of Seqlock

- Execution of Seqlock.



- Get the history: Keep call & return actions only.

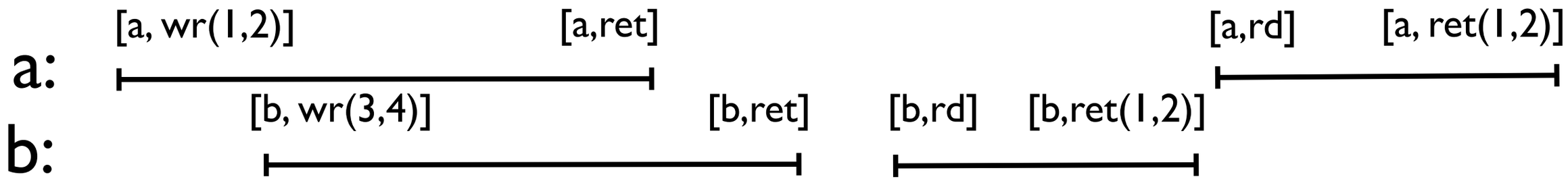
[a,wr(1,2)][b,wr(3,4)][a,ret][b,ret][b,rd][b,ret(1,2)][a,rd][a,ret(1,2)]

  - Find a LinH-related history in [Spec]: Permute actions, while keeping thread & happen-before order.

[b,wr(3,4)][b,ret][a,wr(1,2)][a,ret][b,rd][b,ret(1,2)][a,rd][a,ret(1,2)]

Linearizability of Seqlock

- Execution of Seqlock.



- Get the history: Keep call & return actions only.

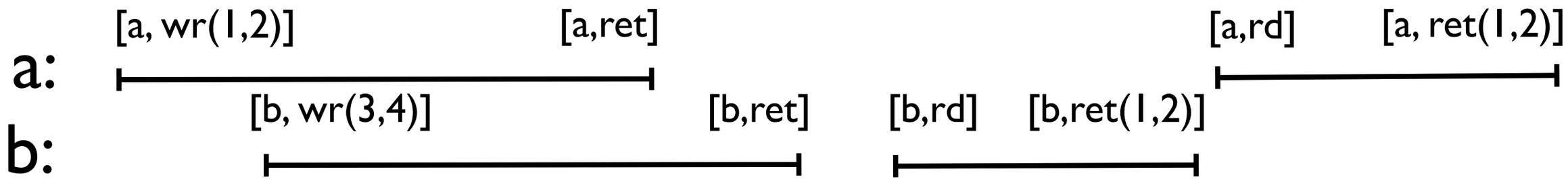
[a,wr(1,2)][b,wr(3,4)][a,ret][b,ret][b,rd][b,ret(1,2)][a,rd][a,ret(1,2)]

  - Find a LinH-related history in [Spec]: Permute actions, while keeping thread & happen-before order.

[b,wr(3,4)][b,ret][a,wr(1,2)][a,ret][b,rd][b,ret(1,2)][a,rd][a,ret(1,2)]

Linearizability of Seqlock

- Execution of Seqlock.



- Get the history: Keep call & return actions only.

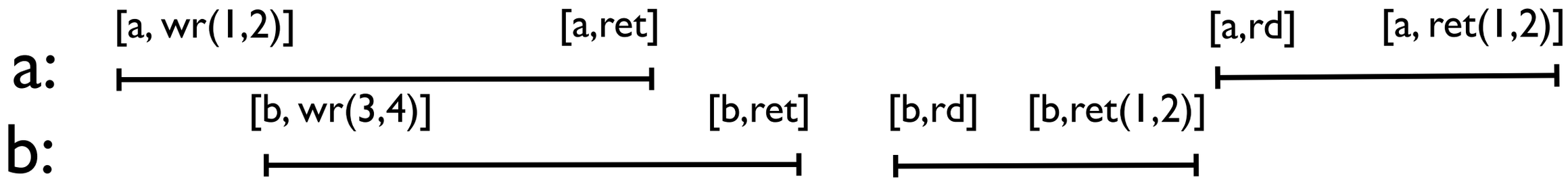
[a,wr(1,2)][b,wr(3,4)][a,ret][b,ret][b,rd][b,ret(1,2)][a,rd][a,ret(1,2)]

  - Find a LinH-related history in [Spec]: Permute actions, while keeping thread & happen-before order.

[b,wr(3,4)][b,ret][a,wr(1,2)][a,ret][b,rd][b,ret(1,2)][a,rd][a,ret(1,2)]

Linearizability of Seqlock

- Execution of Seqlock.



- Get the history: Keep call & return actions only.

[a,wr(1,2)][b,wr(3,4)][a,ret][b,ret][b,rd][b,ret(1,2)][a,rd][a,ret(1,2)]

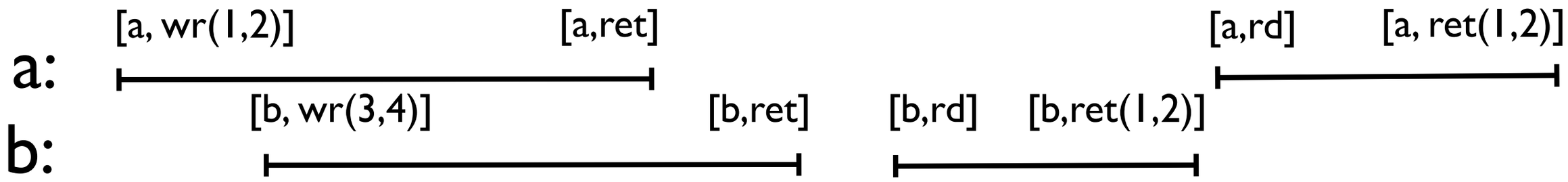
  - Find a LinH-related history in [Spec]: Permute actions, while keeping thread & happen-before order.

[b,wr(3,4)][b,ret][a,wr(1,2)][a,ret][b,rd][b,ret(1,2)][a,rd][a,ret(1,2)]

X [b,wr(3,4)][b,ret][a,wr(1,2)][a,ret][a,rd][a,ret(1,2)][b,rd][b,ret(1,2)]

Linearizability of Seqlock

- Execution of Seqlock.



- Get the history: Keep call & return actions only.

[a,wr(1,2)][b,wr(3,4)][a,ret][b,ret][b,rd][b,ret(1,2)][a,rd][a,ret(1,2)]

  - Find a LinH-related history in [Spec]: Permute actions, while keeping thread & happen-before order.

[b,wr(3,4)][b,ret][a,wr(1,2)][a,ret][b,rd][b,ret(1,2)][a,rd][a,ret(1,2)]

X [b,wr(3,4)][b,ret][a,wr(1,2)][a,ret][a,rd][a,ret(1,2)][b,rd][b,ret(1,2)]

Linearizability

- Binary relation on concurrent libraries, usually from a highly-concurrent one to a spec.
- L1 is linearizable wrt. L2 (denoted L1 $\sqsubseteq$ L2) iff

  $\forall h1 \in [L1].\ \exists h2 \in [L2].\ \ h1\,[LinR]\,h2.$

- h1 [LinR] h2 holds iff h2 is a permutation of h1 s.t.

  1. proj(h1,a) = proj(h2,a) for all thread-ids a, and
  2. the happen-before order of h1 is preserved by h2.

PL perspective

History viewed as abstraction

- mean(h) = { t | ProjectLibraryActions(t) = h }.
- Says all traces whose interactions with the object are h.

mean([a,wr(1,2)][a,ret][b,rd][b,ret(1,2)]) = {

[a,wr(1,2)][a,ret][b,rd][a,x:=8][b,ret(1,2)],

[a,wr(1,2)][b,y:=0][a,ret][b,rd][a,x:=8][b,ret(1,2)],

[a,wr(1,2)][a,ret][a,x:=8][b,assume(x=8)][b,rd][b,ret(1,2)],

...}

- [a,wr(1,2)][a,x:=0][a,ret][b,rd][b,ret(1,2)] is not in the set.

Happen-before in terms of dependency

[Theorem] For all actions i,j of h,

HappenBefore(i,j,h) iff ∃t∈mean(h). Depend(i,j,t).

- Says “HappenBefore” is an abstraction of dependency.

Happen-before in terms of dependency

[Theorem] For all actions i,j of h,

HappenBefore(i,j,h) iff ∃t∈mean(h). Depend(i,j,t).

- Says “HappenBefore” is an abstraction of dependency.
- only-if by example: h = [a,rd][a,ret(0,0)][b,rd][b,ret(0,0)]

V[a,rd][a,ret(0,0)][a,x:=1][b,assume(x=1)][b,rd][b,ret(0,0)]

Happen-before in terms of dependency

[Theorem] For all actions i,j of h,

HappenBefore(i,j,h) iff ∃t∈mean(h). Depend(i,j,t).

- Says "HappenBefore" is an abstraction of dependency.
- only-if by example: h = [a,rd][a,ret(0,0)][b,rd][b,ret(0,0)]

✓ [a,rd][a,ret(0,0)][a,x:=1][b,assume(x=1)][b,rd][b,ret(0,0)]

- if by example: h = [a,rd][b,rd][a,ret(0,0)][b,ret(0,0)]

✗ [a,rd][b,rd][a,ret(0,0)][a,x:=1][b,assume(x=1)][b,ret(0,0)]

Trace equivalence

- t1 ~ t2 iff t2 can be obtained from t1 by swapping independent actions from the client perspective.
- [a,x:=4][b,y:=11] ~ [b,y:=11][a,x:=4]
- [a,x:=4][b,x:=11] $\not\sim$ [b,x:=11][a,x:=4]
- [a,x:=4][a,y:=11] $\not\sim$ [a,y:=11][a,x:=4]
- [a,wr(1,2)][a,ret][b,wr(3,4)] ~ [a,wr(1,2)][b,wr(3,4)][a,ret]

Strong rearrangement lemma

[Lemma]

$h1[\mathrm{LinR}]h2 \;\; \text{iff} \;\; \forall t1 \in \mathrm{mean}(h1).\ \exists t2 \in \mathrm{mean}(h2).\ t1 \sim t2.$

- Says 1) mean(h1) is a subset of mean(h2) in a sense, and 2) we can always replace h2 by h1.

- only-if by example:

h1 = [a,rd][b,rd][b,ret(0,0)][a,ret(0,0)]

t1 = [a,rd][b,rd][b,ret(0,0)][b,x:=1][a,ret(0,0)]

h2 = [a,rd][a,ret(0,0)][b,rd][b,ret(0,0)]

Strong rearrangement lemma

[Lemma]

$h1[\mathrm{LinR}]h2 \;\; \text{iff} \;\; \forall t1 \in \mathrm{mean}(h1).\ \exists t2 \in \mathrm{mean}(h2).\ t1 \sim t2.$

- Says 1) mean(h1) is a subset of mean(h2) in a sense, and 2) we can always replace h2 by h1.

- only-if by example:

h1 = [a,rd][b,rd][b,ret(0,0)][a,ret(0,0)]

t1 = [a,rd][b,rd][b,ret(0,0)][b,x:=1][a,ret(0,0)]

h2 = [a,rd][a,ret(0,0)][b,rd][b,ret(0,0)]

t2 = [a,rd][a,ret(0,0)][b,rd][b,ret(0,0)][b,x:=1]

Strong rearrangement lemma

[Lemma]

$$h1[LinR]h2 \quad \text{iff} \quad \forall t1 \in mean(h1).\ \exists t2 \in mean(h2).\ t1 \sim t2.$$

- Says 1) mean(h1) is a subset of mean(h2) in a sense, and 2) we can always replace h2 by h1.

- if by example:

h1=[b,rd][b,ret(0,0)][a,rd][a,ret(0,0)]

t1=[b,rd][b,ret(0,0)][b,x:=1][a,assume(x=1)][a,rd][a,ret(0,0)]

h2=[a,rd][a,ret(0,0)][b,rd][b,ret(0,0)]

Strong rearrangement lemma

[Lemma]

$h1[LinR]h2 \quad iff \quad \forall t1 \in mean(h1).\ \exists t2 \in mean(h2).\ t1 \sim t2.$

- Says 1) mean(h1) is a subset of mean(h2) in a sense, and 2) we can always replace h2 by h1.

- if by example:

h1=[b,rd][b,ret(0,0)][a,rd][a,ret(0,0)]

t1=[b,rd][b,ret(0,0)][b,x:=1][a,assume(x=1)][a,rd][a,ret(0,0)]

h2=[a,rd][a,ret(0,0)][b,rd][b,ret(0,0)]

t2=[a,assume(x=1)][a,rd][a,ret(0,0)][b,rd][b,ret(0,0)][b,x:=1]

Strong rearrangement lemma

[Lemma]

h1[LinR]h2 iff ∀t1∈mean(h1). ∃t2∈mean(h2). t1~t2.

[Corollary]

If we ignore the termination issue, linearizability is equivalent to observational refinement.

(Weak) rearrangement lemma

[Lemma]

h1[LinR]h2 implies $\forall t1 \in mean(h1).\ \exists t2 \in mean(h2).\ t1 \sim t2$.

[Corollaries]

1. If we ignore the termination issue, linearizability implies observational refinement.

2. We can show that L1[L2] $\sqsubseteq$ S1[S2] holds, by proving that L2 $\sqsubseteq$ S2 and L1[S2] $\sqsubseteq$ S1[S2].

Something to remember

1. Histories record client-relevant actions.

2. Happen-before is an abstraction of the dependency of possible client actions.

3. LinR implies trace inclusion modulo trace equivalence.

Exercise 1: liveness

Histories

- So far, I assumed finite histories.
- Embrace infinite histories.
- Histories can include [a,starve], which means that thread a is not scheduled forever.

  [a,wr(1,2)][b,rd][b,ret(0,0)][a,starve]

  [a,wr(1,2)][b,rd][b,ret(0,0)]

  [a,wr(1,2)][b,rd][b,ret(0,0)][b,rd][b,ret(1,2)]...

Happen-before order

h1 [LinR]h2 holds iff h2 is a permutation of h1 s.t.

1. proj(h1,a) = proj(h2,a) for all thread-ids a, and
2. the happen-before order of h1 is preserved by h2.

[Q] What should we include in the happen-before order?

1. h1 = ...[a,starve][b,wr(1,2)]...
2. h1 = ...[a,starve][b,ret]...
3. h1 = ...[a,starve][b,starve]...
4. h1 = ...[a,wr(1,2)][b,starve]...
5. h1 = ...[a,ret][b,starve]...

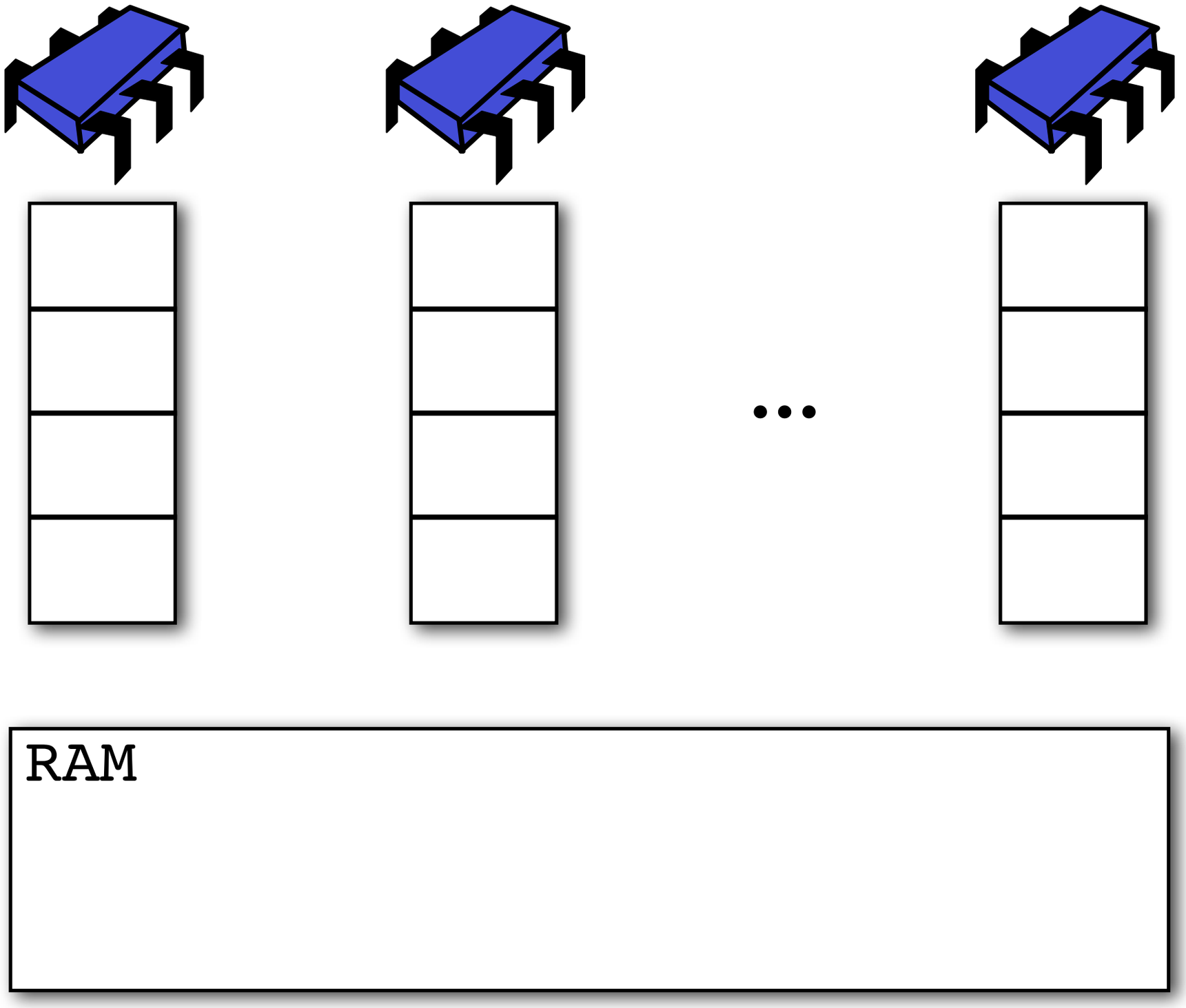
Consequences

- We have the weak rearrangement lemma (after slightly adjustment).
- Linearizability implies observational refinement.
- We can prove that L1[L2] is lock-free by showing the following properties:

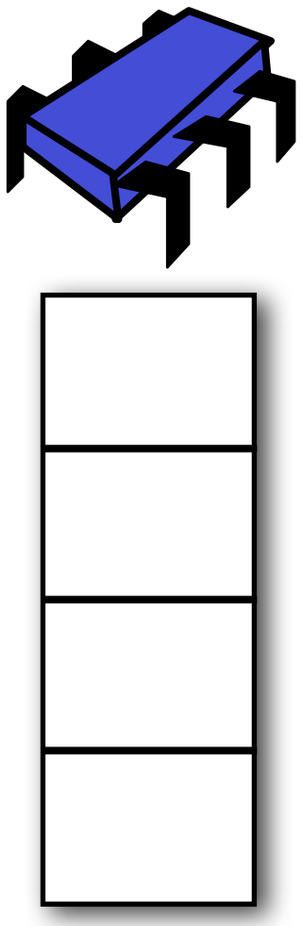
1. L2 is linearizable wrt. S2.
2. L1[S2] is lock-free when all methods of S2 terminates.

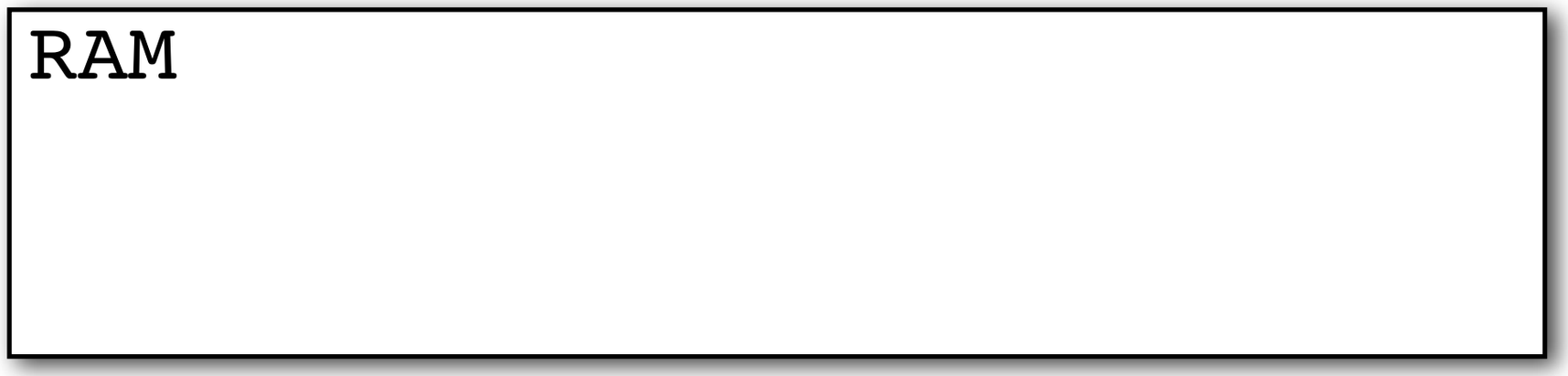
Exercise 2: TSO weak memory model

The TSO memory model (x86)

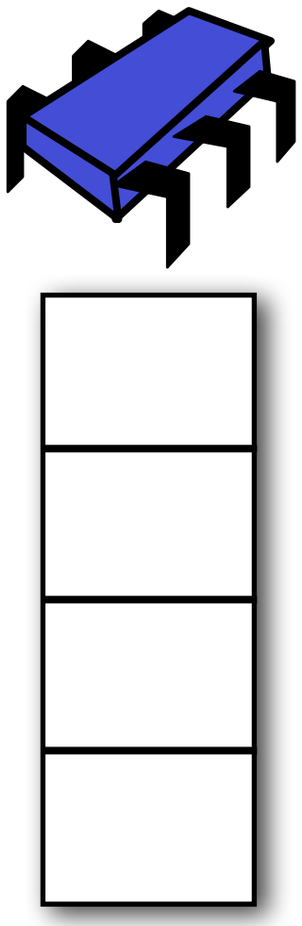


The TSO memory model (x86)





The TSO memory model (x86)

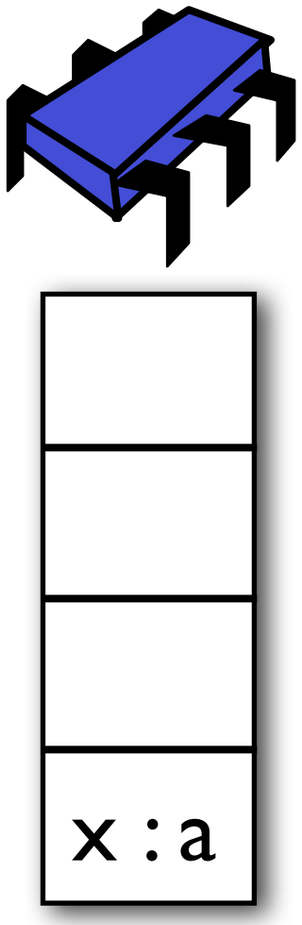


```
f() {
  *x = a;
  *y = b;
  *z = c;
}
```

Writes stored into the write buffer in the order of issue

RAM

The TSO memory model (x86)

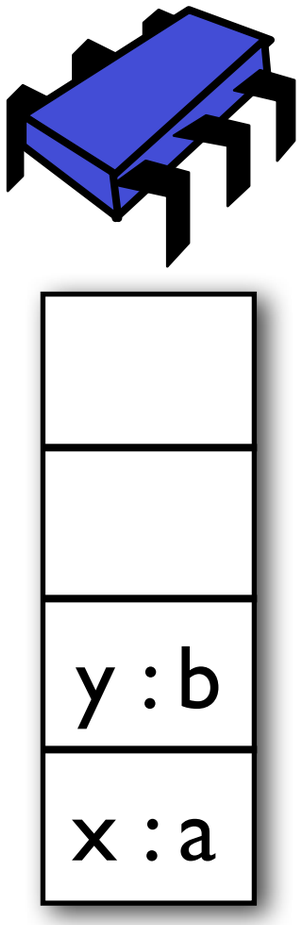



```
f() {
  *x = a;
  *y = b;
  *z = c;
}
```

Writes stored into the write buffer in the order of issue

RAM

The TSO memory model (x86)

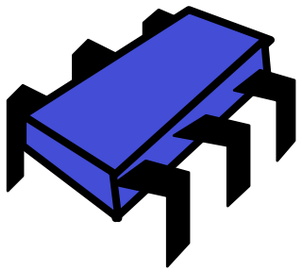


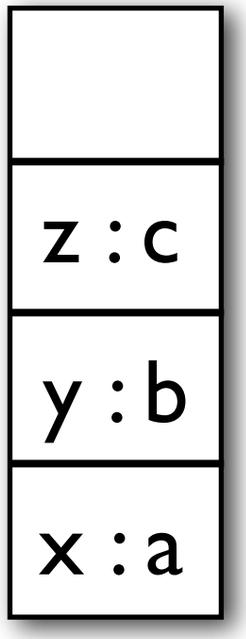

```
f() {
  *x = a;
  *y = b;
  *z = c;
}
```

Writes stored into the write buffer in the order of issue

RAM

The TSO memory model (x86)



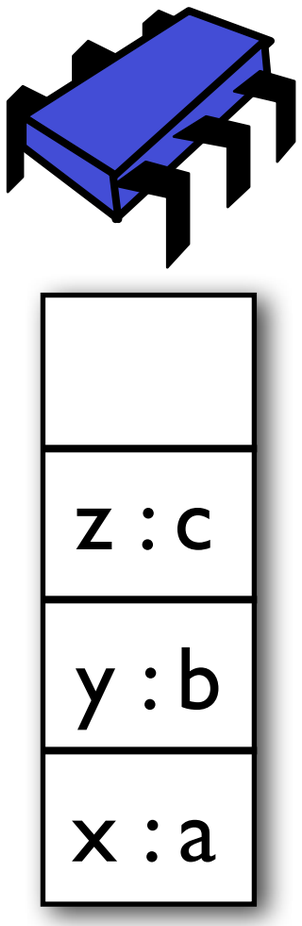



```
f() {
  *x = a;
  *y = b;
  *z = c;
}
```

Writes stored into the write buffer in the order of issue

RAM

The TSO memory model (x86)

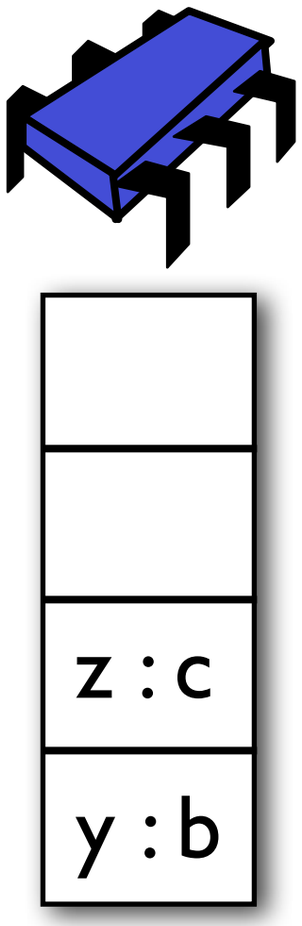



```
f() {
  *x = a;
  *y = b;
  *z = c;
}
```

Writes flushed in FIFO order

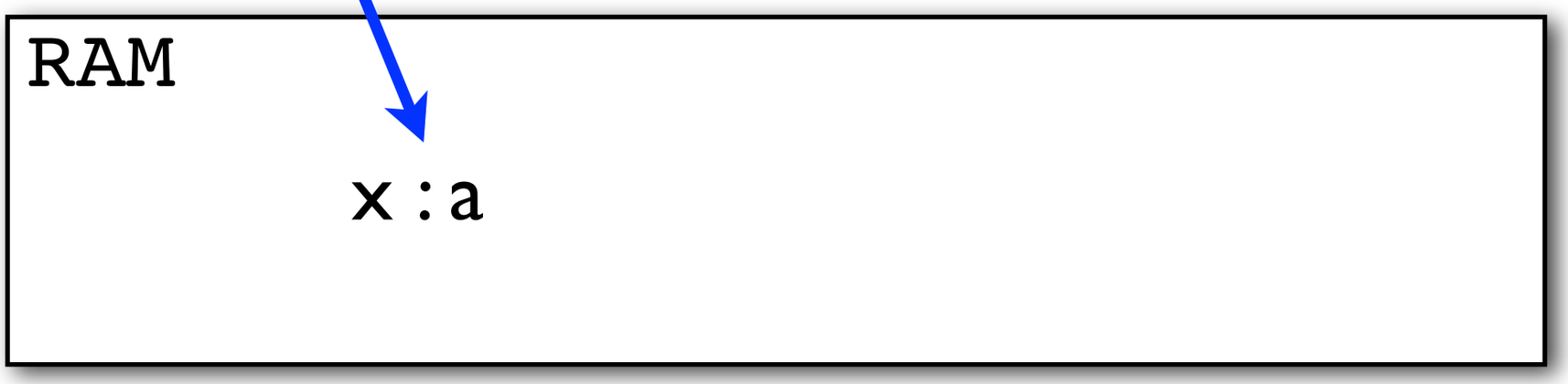
RAM

The TSO memory model (x86)

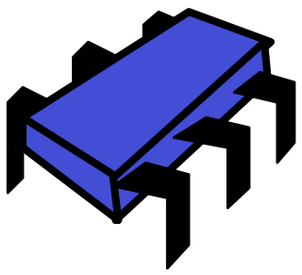



```
f() {
  *x = a;
  *y = b;
  *z = c;
}
```

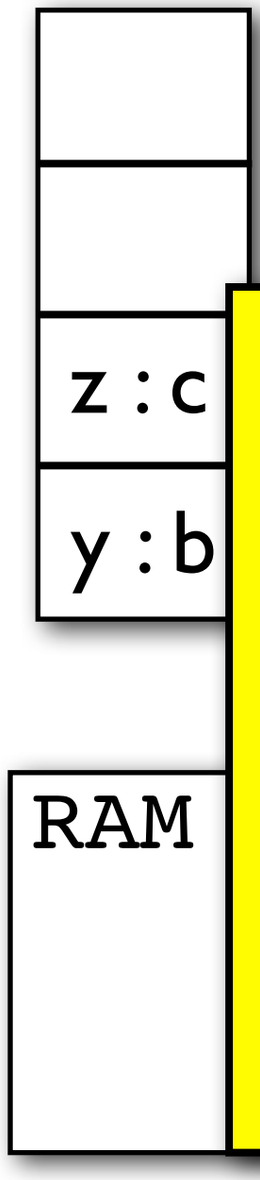
Writes flushed in FIFO order



The TSO memory model (x86)



```
f() {
  *x = a;
  *y = b;
  *z = c;
}
```



- Can be flushed after f returns
- Library behaviour depends on the time of the flush
- Just parameters and return values not enough

Histories

- Include [a,flushS] and [a,flushE].
- They mark the beginning and end of flushing writes issued during method calls.

[a,wr(1,2)][a,ret][a,flushS][b,rd][a,flushE][b,ret(1,2)]

[a,wr(1,2)][b,rd][b,ret(0,0)][a,ret][a,flushS][a,flushE]

Histories

- Include [a,flushS] and [a,flushE].
- They mark the beginning and end of flushing writes issued during method calls.

[a,wr(1,2)][a,ret][a,flushS][b,rd][a,flushE][b,ret(1,2)]

[a,wr(1,2)][b,rd][b,ret(0,0)][a,ret][a,flushS][a,flushE]

[a,wr(1,2)][b,rd][b,ret(1,2)][a,ret][a,flushS][a,flushE]

Happen-before order

[Q] What should we include in the happen-before order?

1. h1 = ...[a,flushE][b,wr(1,2)]...
2. h1 = ...[a,flushE][b,ret]...
3. h1 = ...[a,flushE][b,flushS]...
4. h1 = ...[a,flushE][b,flushE]...
5. h1 = ...[a,flushS][b,wr(1,2)]...
6. h1 = ...[a,flushS][b,ret], and 2 other possibilities.

Consequences

- Again, we get a version of the weak rearrangement lemma.
- Linearizability implies observational refinement.

Tips for obtaining a sensible correctness condition

- Identify the type of client programs that you have in mind.
- Define histories, which keep information of library executions relevant to client programs.
- Identify properties of histories that client programs observe about histories.
- Such properties can be expressed by HB order, which capture client-side dependency.