Two Levels of Reuse for Proving Correctness of Concurrent Type Systems

John Boyland
Chao Sun
Yang Zhao, Bill Retert

ETH Zürich
University of Wisconsin-Milwaukee

Software Correctness & Reliability

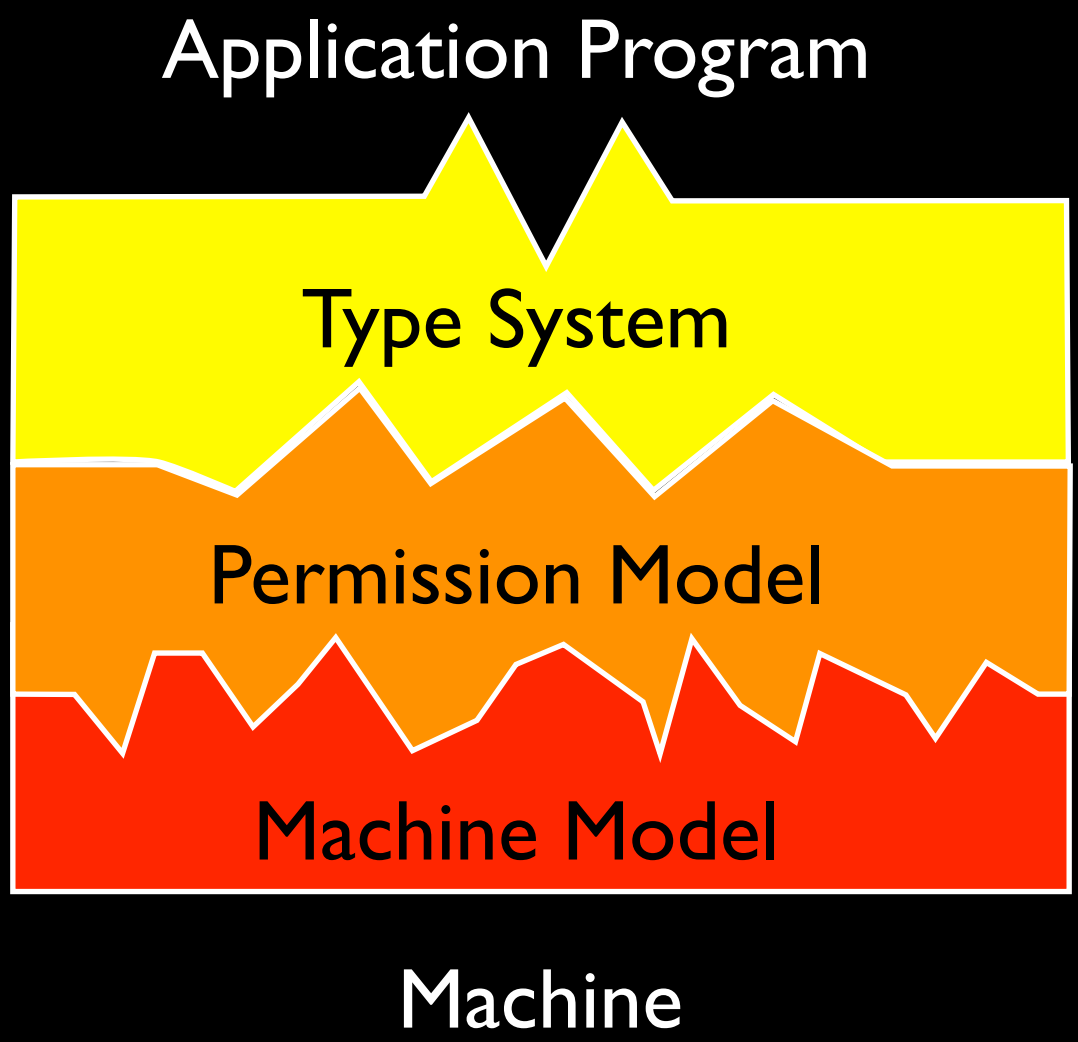
Summary: Level #1

- Type systems give rules to follow;
- Showing rules imply correctness is hard;
- Give semantics of rules in low-level system;
- Show that following rules is sound at low-level.

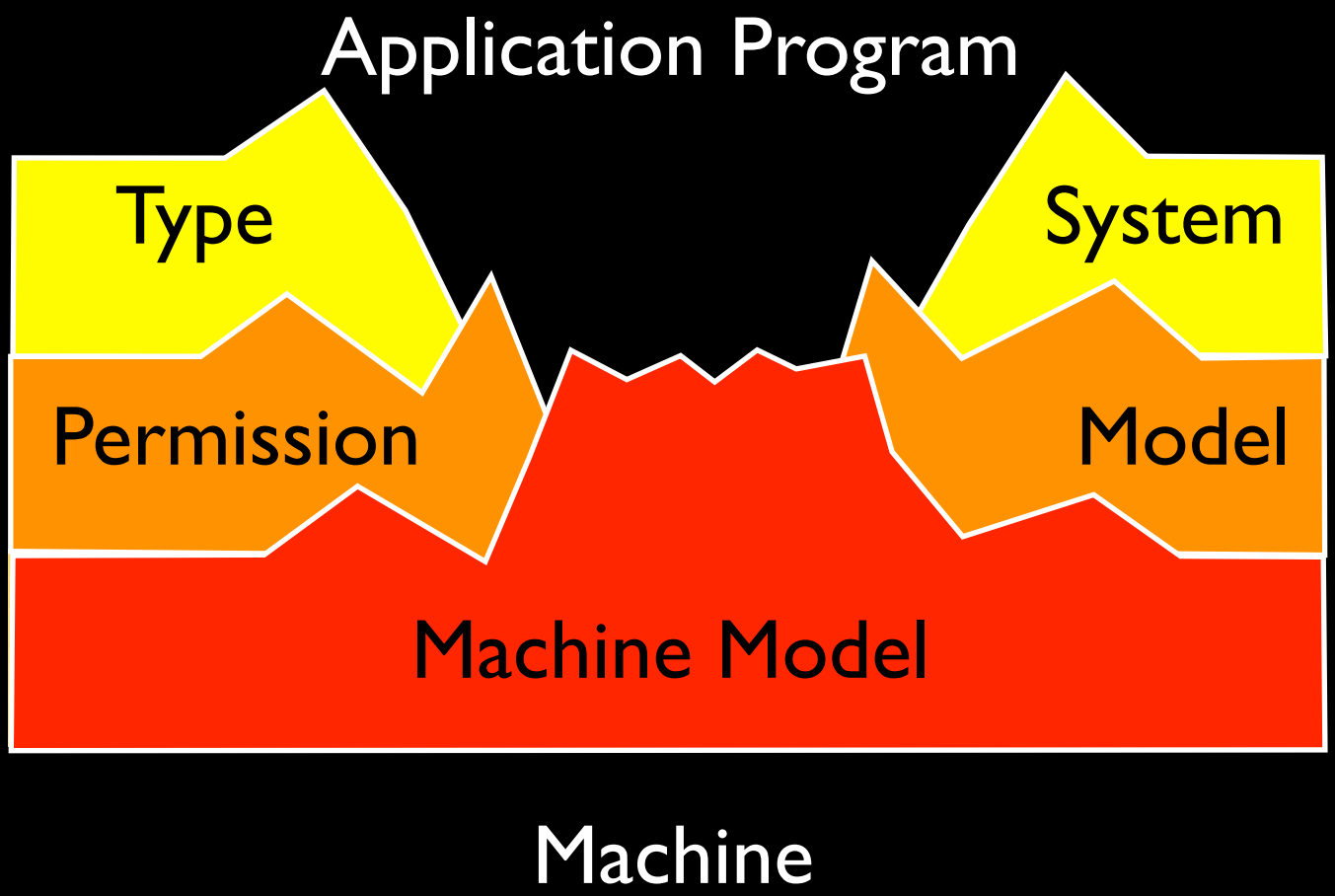
Summary: Level #2

- Reasoning about concurrency is hard.
- The “happens-before” relationship involves events from all threads
- Provide model that encodes “happens-before” in instruction execution.
- Prove that system follows this model.

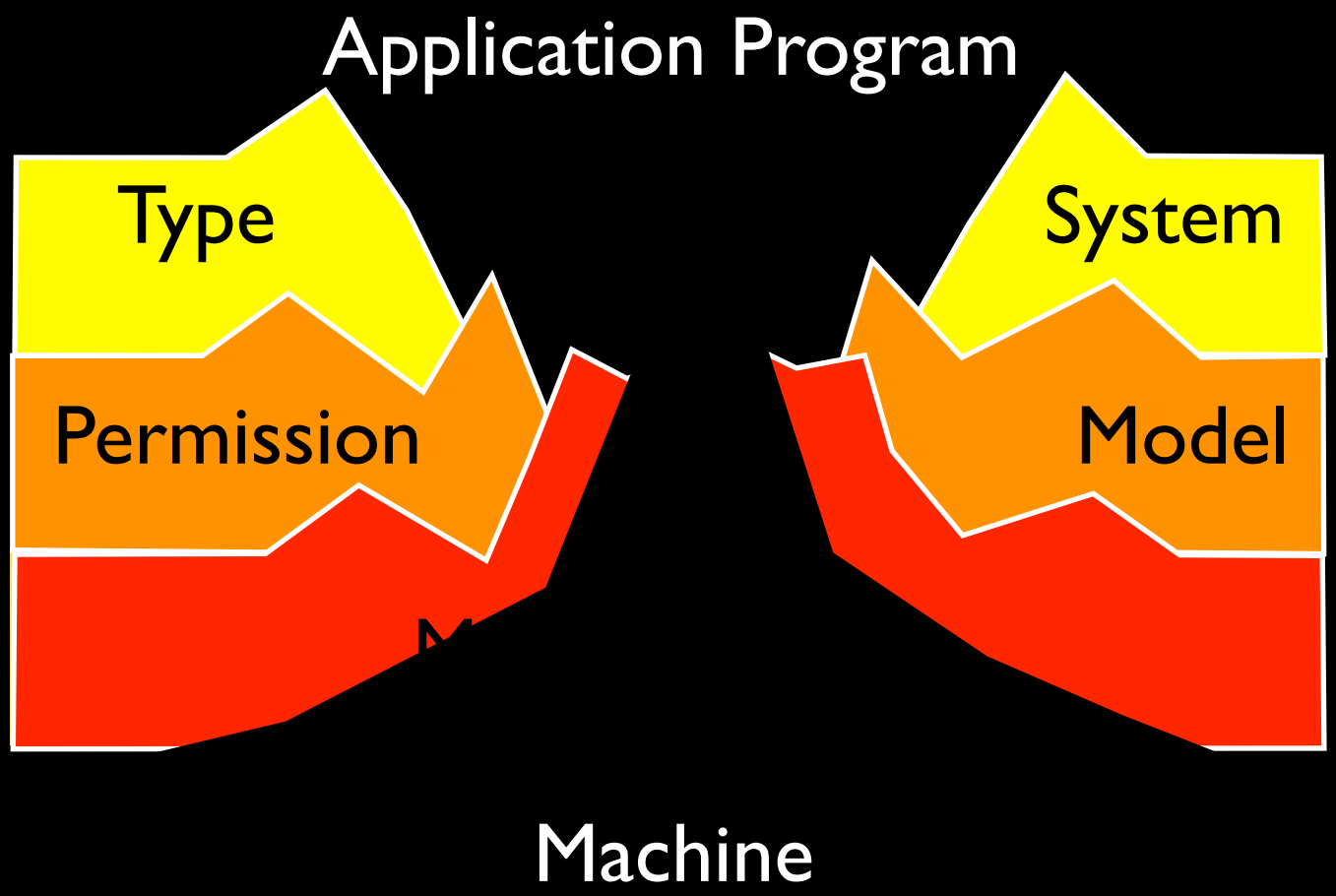
Ideal Proof Structure



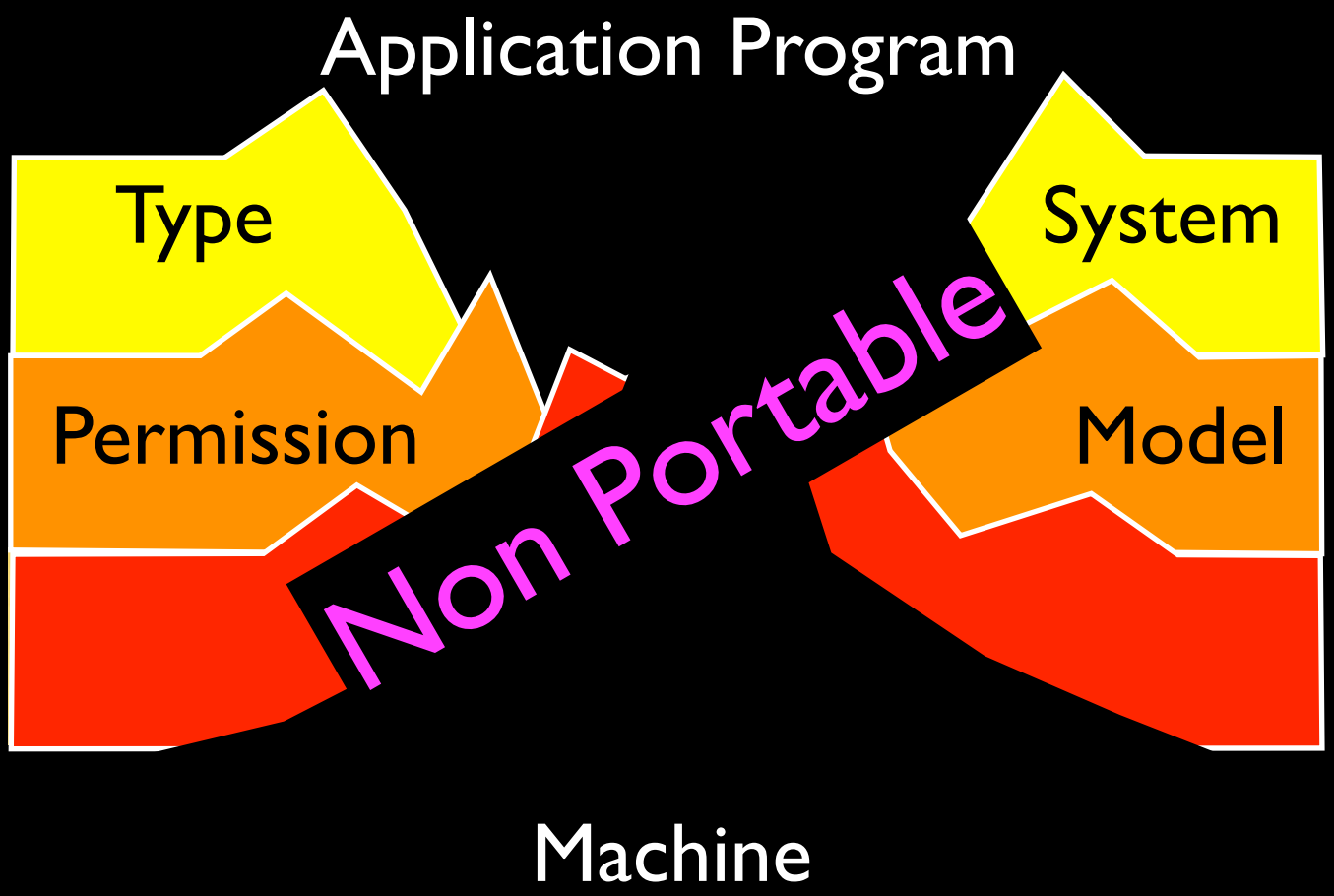
Less Than Ideal



Less Than Ideal



Less Than Ideal



Example: Level #1

- Fractional Permissions with Nesting
  1. Access (fraction = read)
  2. Hierarchy (nested = owned)
  3. Invariants

  ➡ Very low-level
- Hide with a type system

Low-Level Perm. Model

- Permission Model

1. Access to field requires permission.
2. Synchronization on p (where p.All < 0.Lck)
   - requires lock order, provides p.All, OR
   - requires p.All, provides p.All
3. Writing a volatile field requires invariant; reading it provides invariant.

Low-Level Perm. Model

- Permission model does *not* specify
  1. What is nested in p.All
  2. Volatile invariant $I_f(r)$
  3. Function(method) signatures
  4. Types/subtypes/inheritance
- But if you follow rules, program won't go "wrong" (bad field access / race / deadlock).

Type Systems

- Many concepts: ownership, uniqueness, immutable, read-only, unique-write, guarded, raw vs. initialized, non-null, abstract read permissions, etc.
- All are given semantics in permission logic.
- Type system rules translate into permission transformation. Soundness w.r.t. permissions.

Example: Non-Null

- Non-null system:
  1. some fields annotated @NonNull
  2. constructor has a special role to play
  3. object moves from “raw” to “initialized”
- Concepts translated to permissions.
- Proof: If a program is non-null typesafe, then its translation is permission safe.

Example: Non-Null

- Non-null system:
  1. some fields annotated @NonNull
  2. constructor has a special role to play
  3. object moves from “raw” to “initialized”
- Concepts translated to permissions.
- Proof: If a program is non-null typesafe, then its translation is permission safe.

Mechanization in Twelf by Chao Sun

Related Work #1

- Boogie Methodology
  1. High-level concepts translated down
     - "Verification Condition Generator"
  2. Verification done at lower-level
     - Some "leakage" up to programmer if assistance needed.

Related Work #1

- SIL ("Semper Intermediate Language")
  1. Automatic translation from Chalice/Scala
  2. Symb. execution / Abstract Interpretation
     - Again, problems can "leak" back up.

- Others!

Example: Level #2

- Volatility is useful and non-trivial, but often omitted from lightweight execution models.
- We include `volatile` and use "write keys" to track "happens-before" at "run-time";
- We can model JMM-inspired "correct synchronization."

How to Prove Safety?

Previous Way:

1. Define execution semantics;
2. Define type system;
3. Prove subject reduction (soundness);
4. Prove that type system avoids races.

How to Prove Safety?

Previous Way:

1. Define execution semantics;
2. Define type system;
3. Prove subject reduction (soundness);
4. Prove that type system avoids races.

Current semantics omit volatile

How to Prove Safety?

Previous Way:

1. Define execution semantics;
2. Define type system;
3. Prove subject reduction (soundness);
4. Prove that type system avoids races.

Current semantics omit volatile

Complex proof using global reasoning

How to Prove Safety!

New way:

1. *Define execution semantics*; (DONE)
2. Define type system;
3. Prove subject reduction (soundness);

~~*4. Prove that type system avoids races.*~~

How to Prove Safety!

New way:

No direct thread interaction

1. *Define execution semantics*; (DONE)
2. Define type system;
3. Prove subject reduction (soundness);
4. ~~*Prove that type system avoids races.*~~

"Correctly Synchronized"

A program is *correctly synchronized* if and only if in all sequentially consistent executions, all conflicting accesses [RW,WR,WW] to non-volatile variables are ordered by "happens-before" edges. [JMM = Java Memory Model]

- Only correctly synchronized programs can rely on sequential consistency.

“Happens Before”

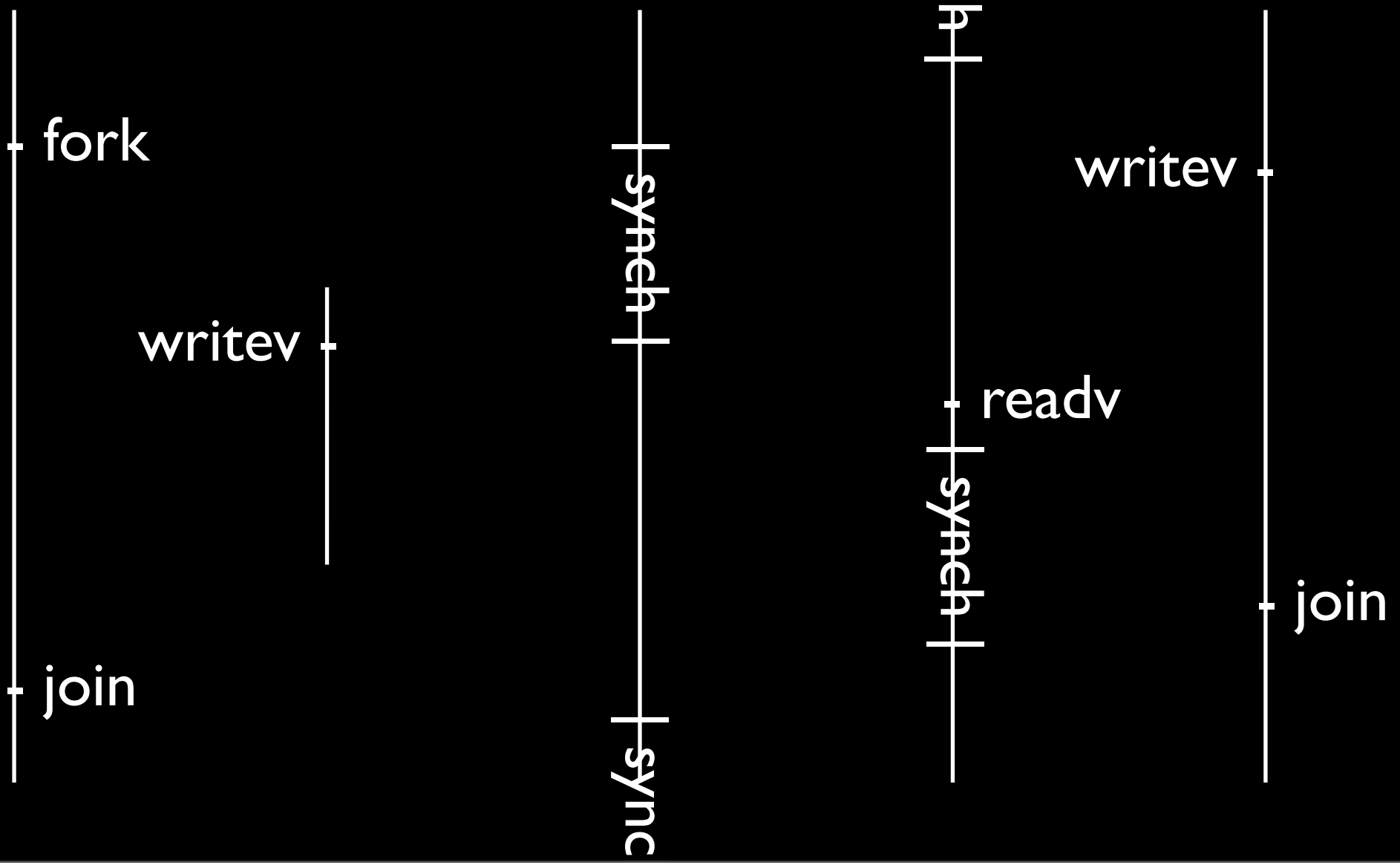
- Intra-thread program order PLUS “synchronizes with” edges:
  1. `fork` to first instruction in thread;
  2. last instruction in thread to `join`;
  3. release lock to acquire lock;
  4. volatile write to volatile read.

“Happens Before”

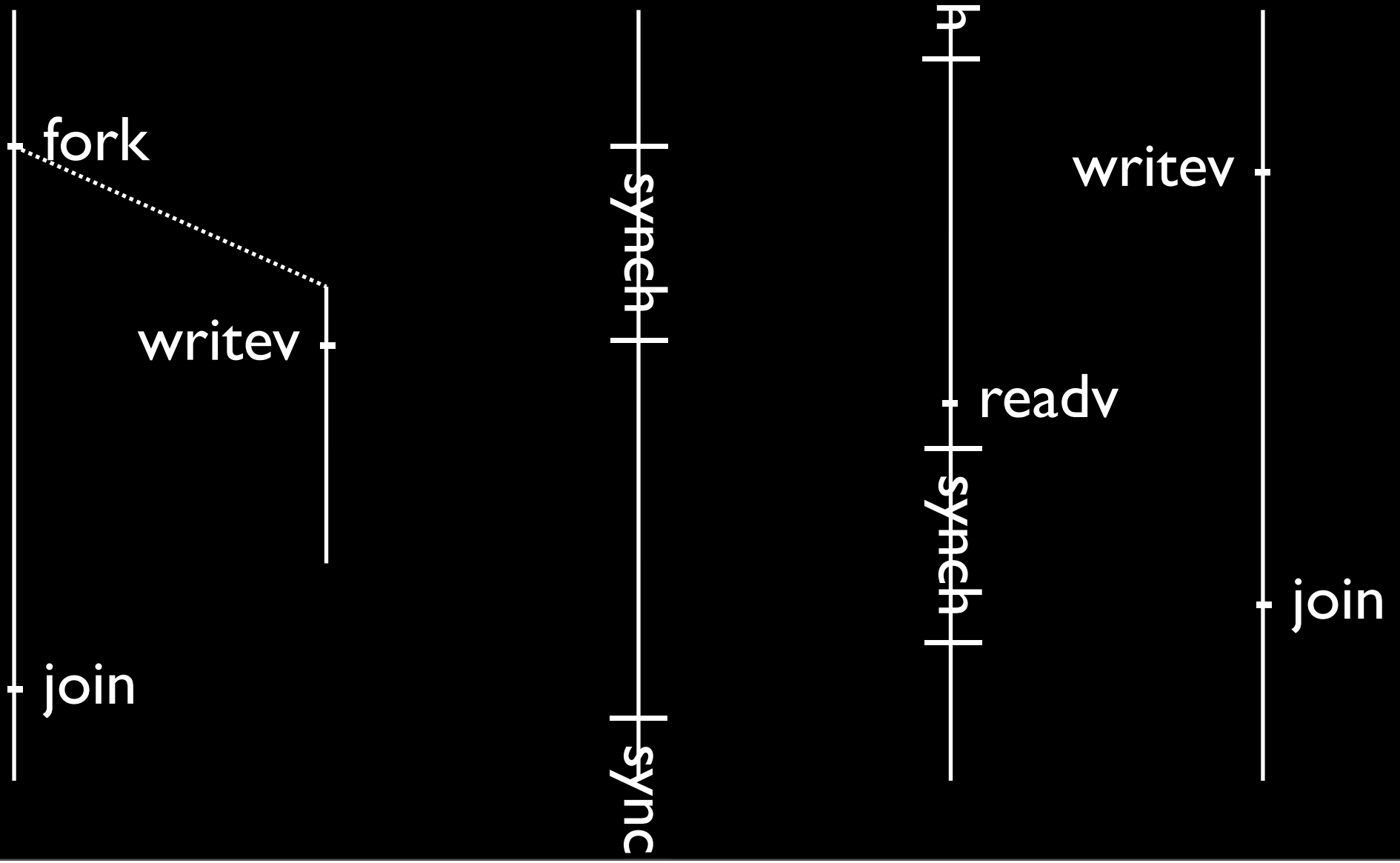
- Intra-thread program order PLUS “synchronizes with” edges:
  1. `fork` to first instruction in thread;
  2. last instruction in thread to `join`;
  3. release lock to acquire lock;
  4. volatile write to volatile read.

Volatile cannot be ignored!

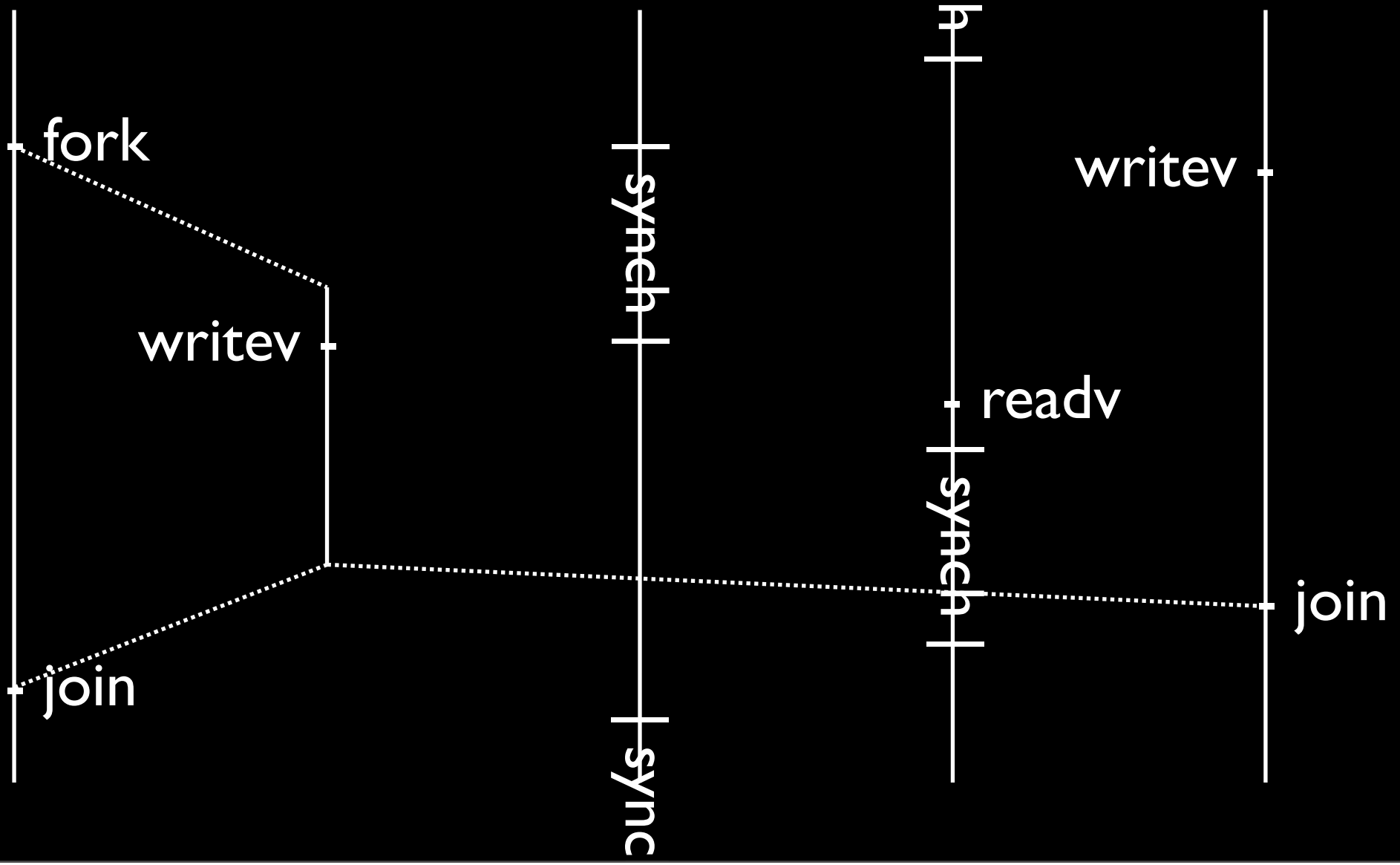
Example



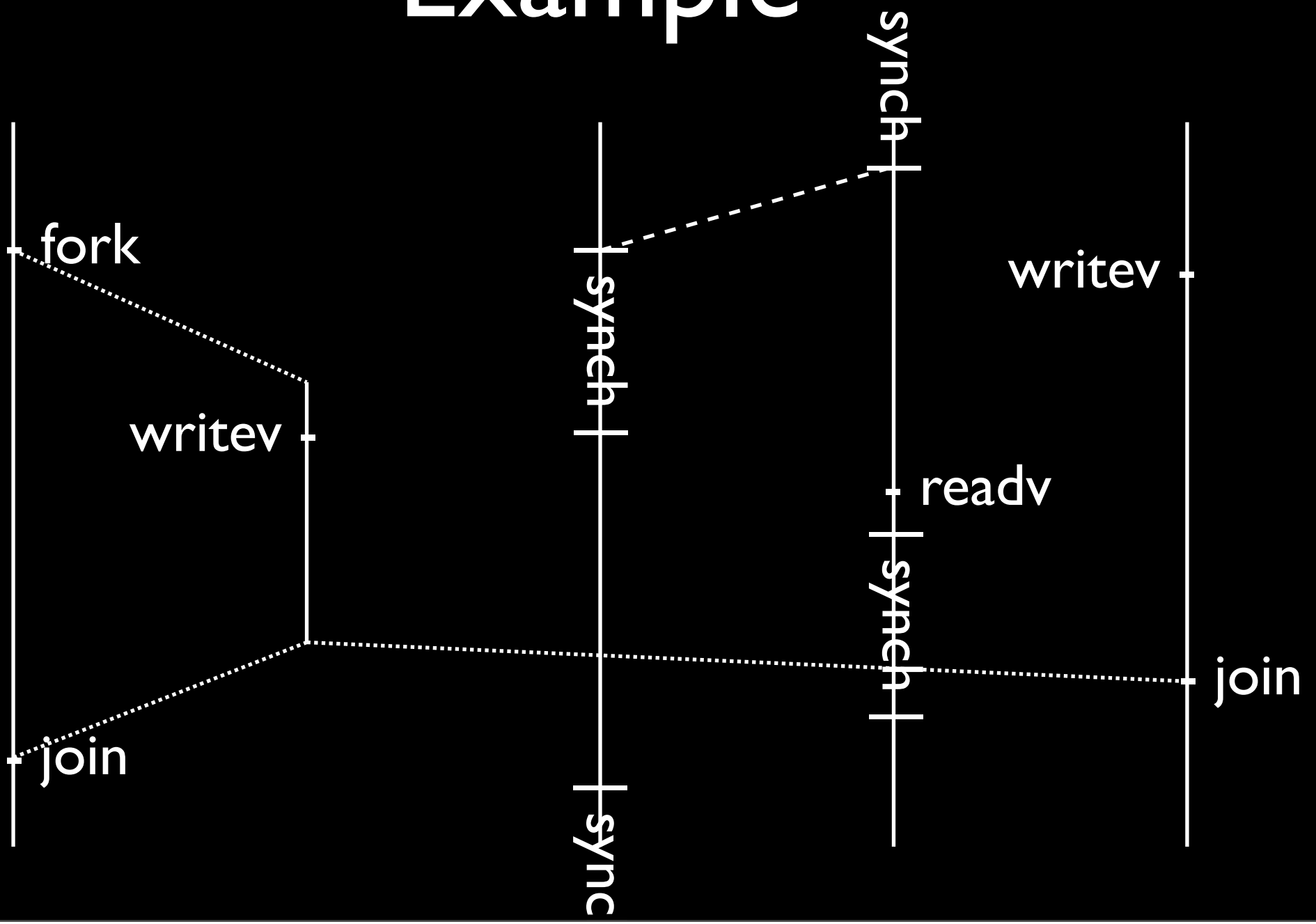
Example



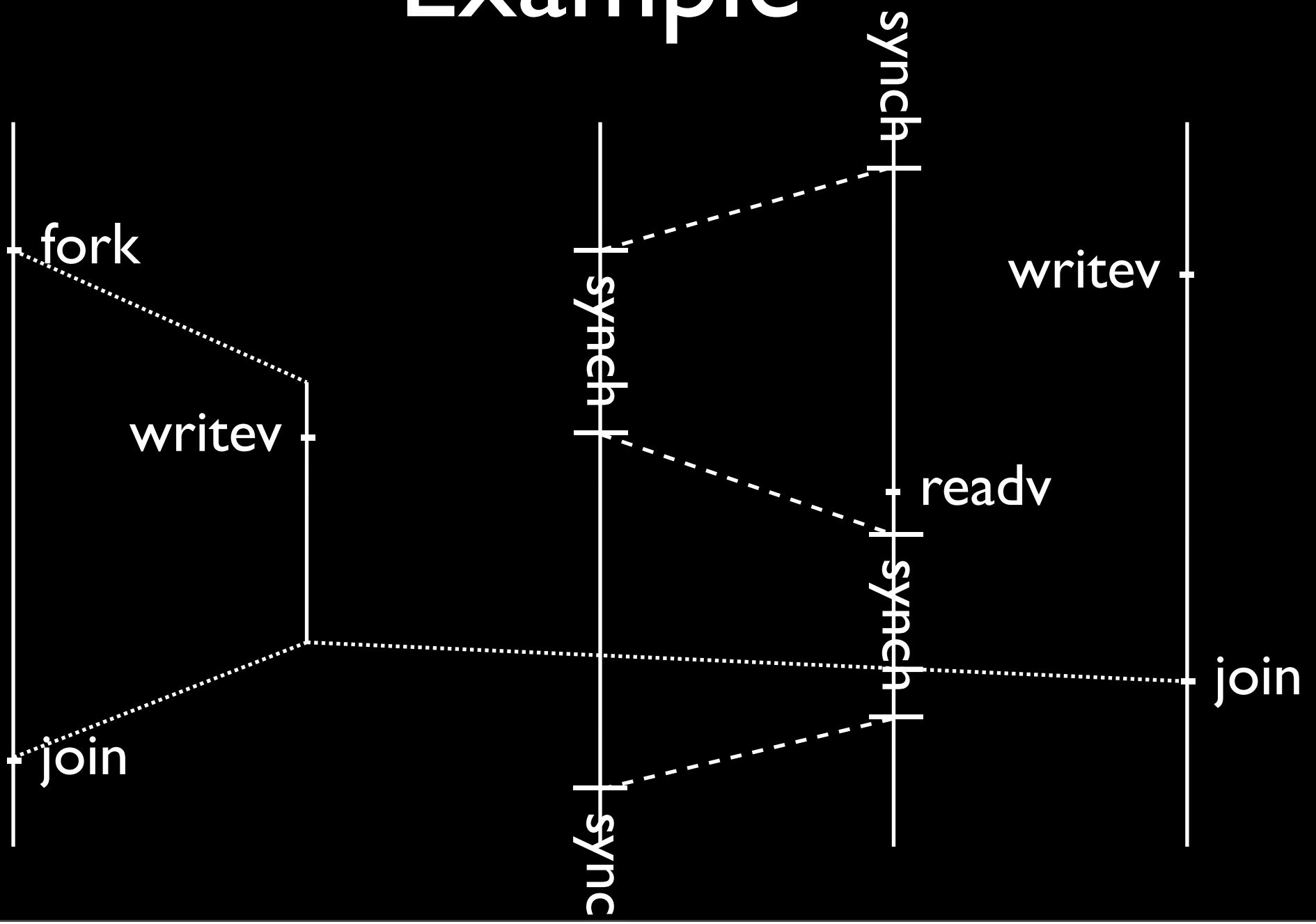
Example



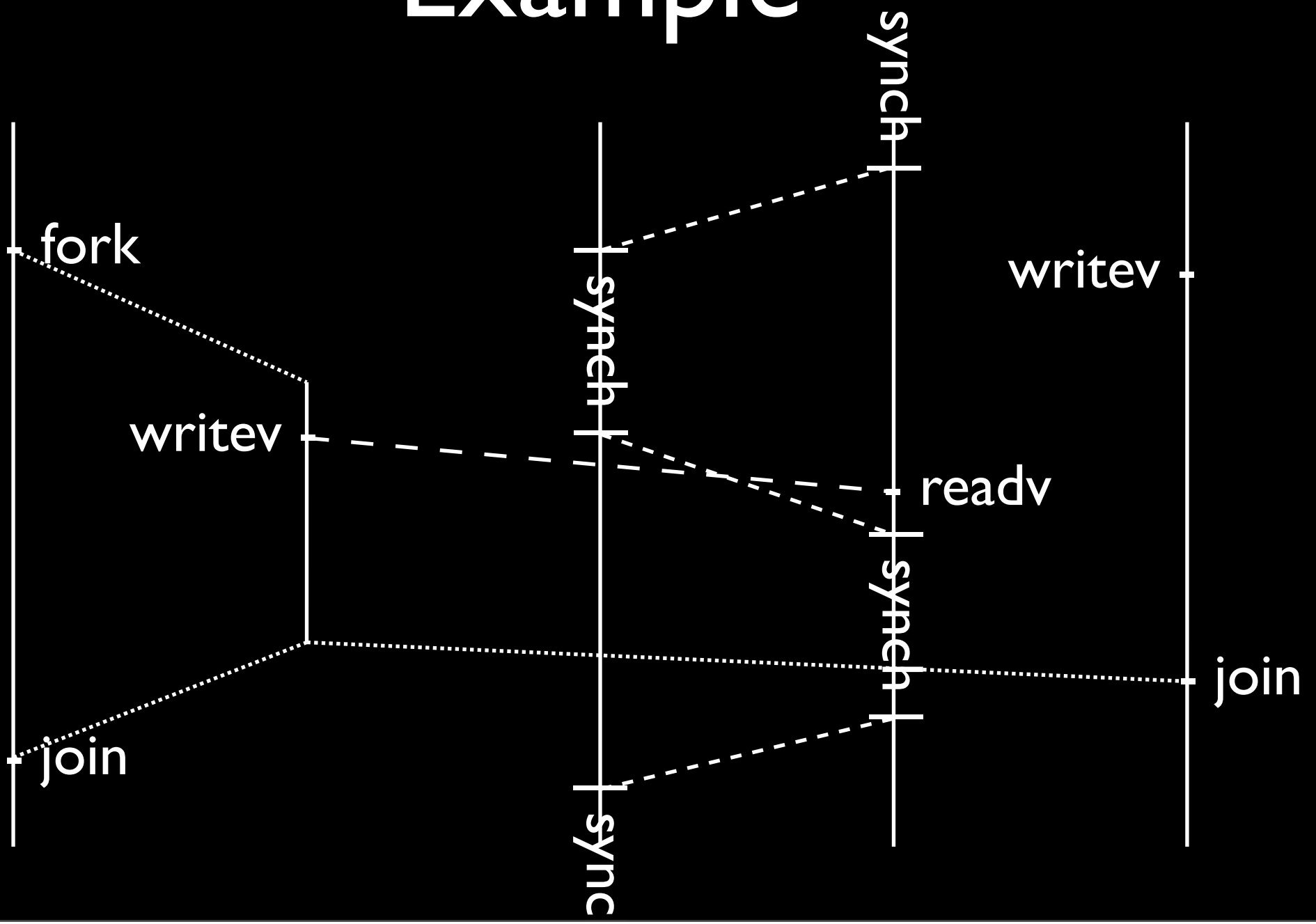
Example



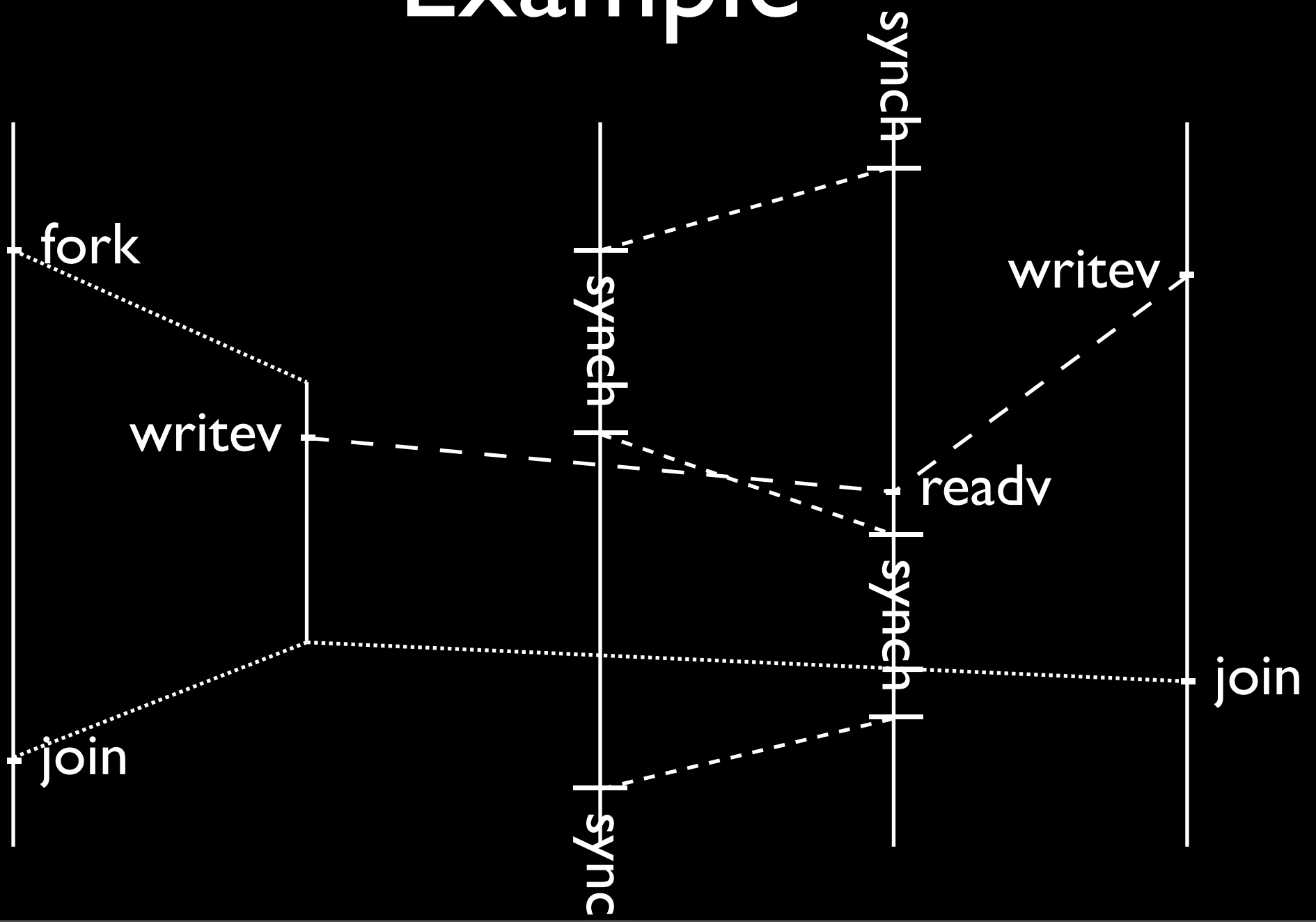
Example



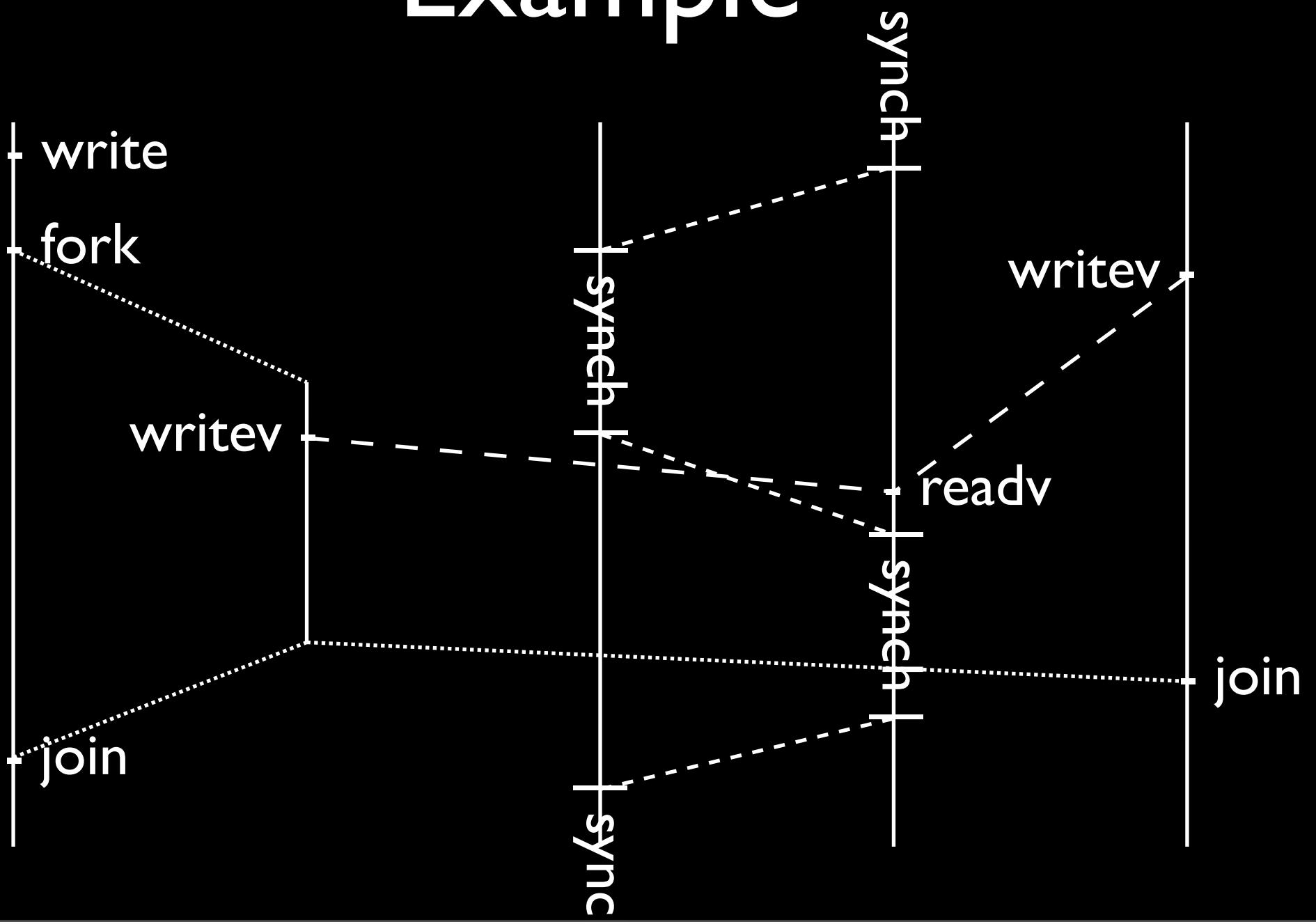
Example



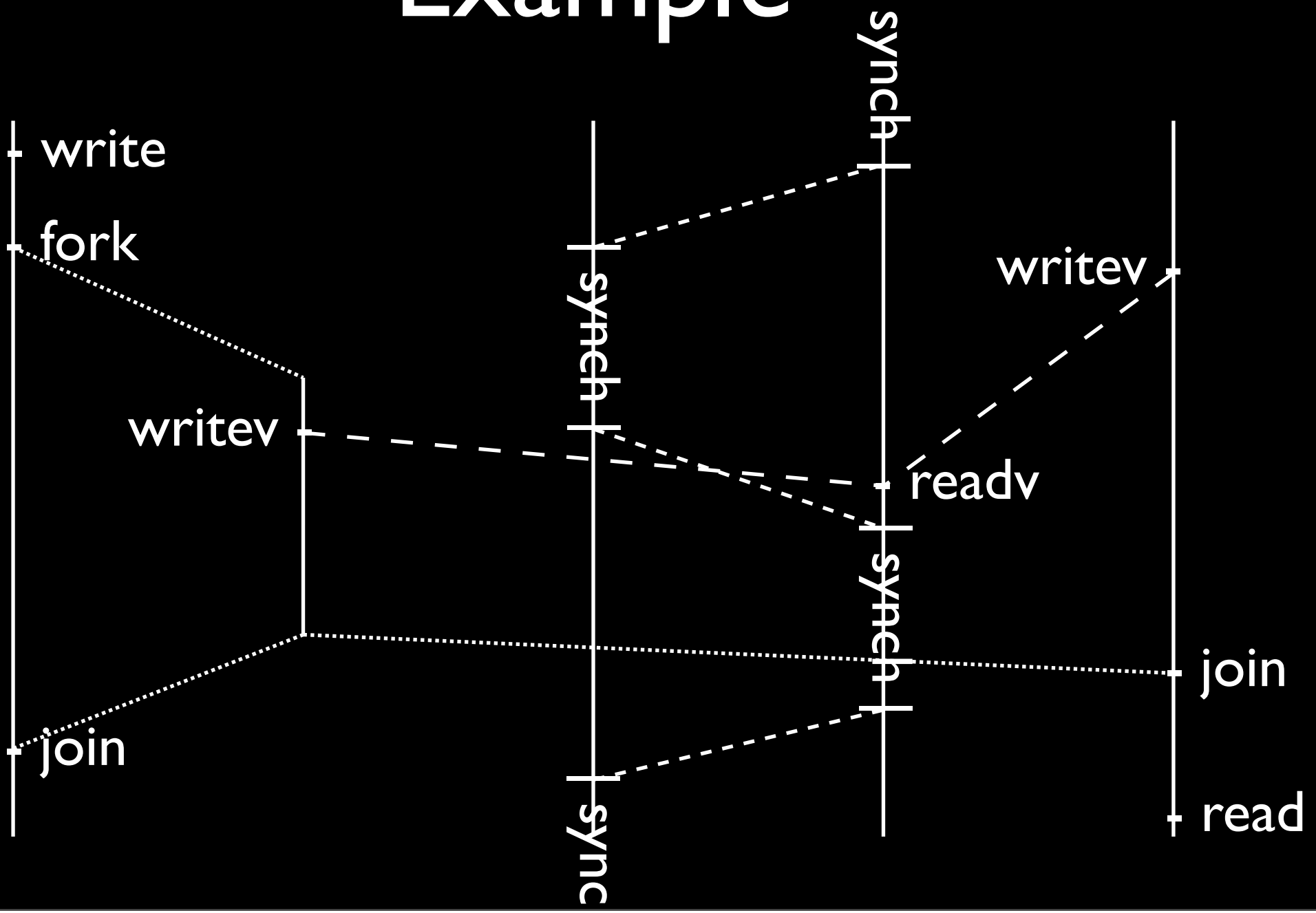
Example



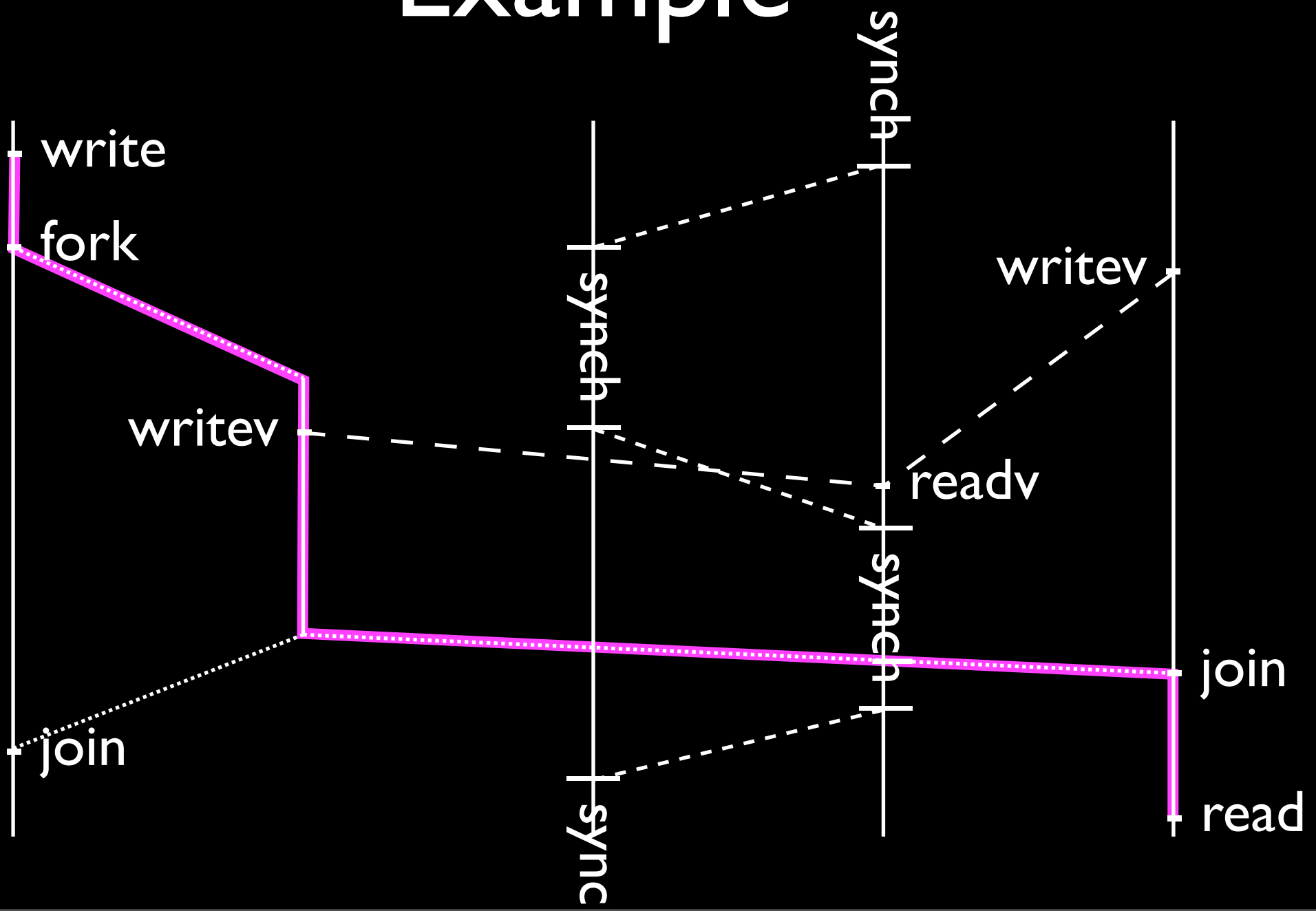
Example



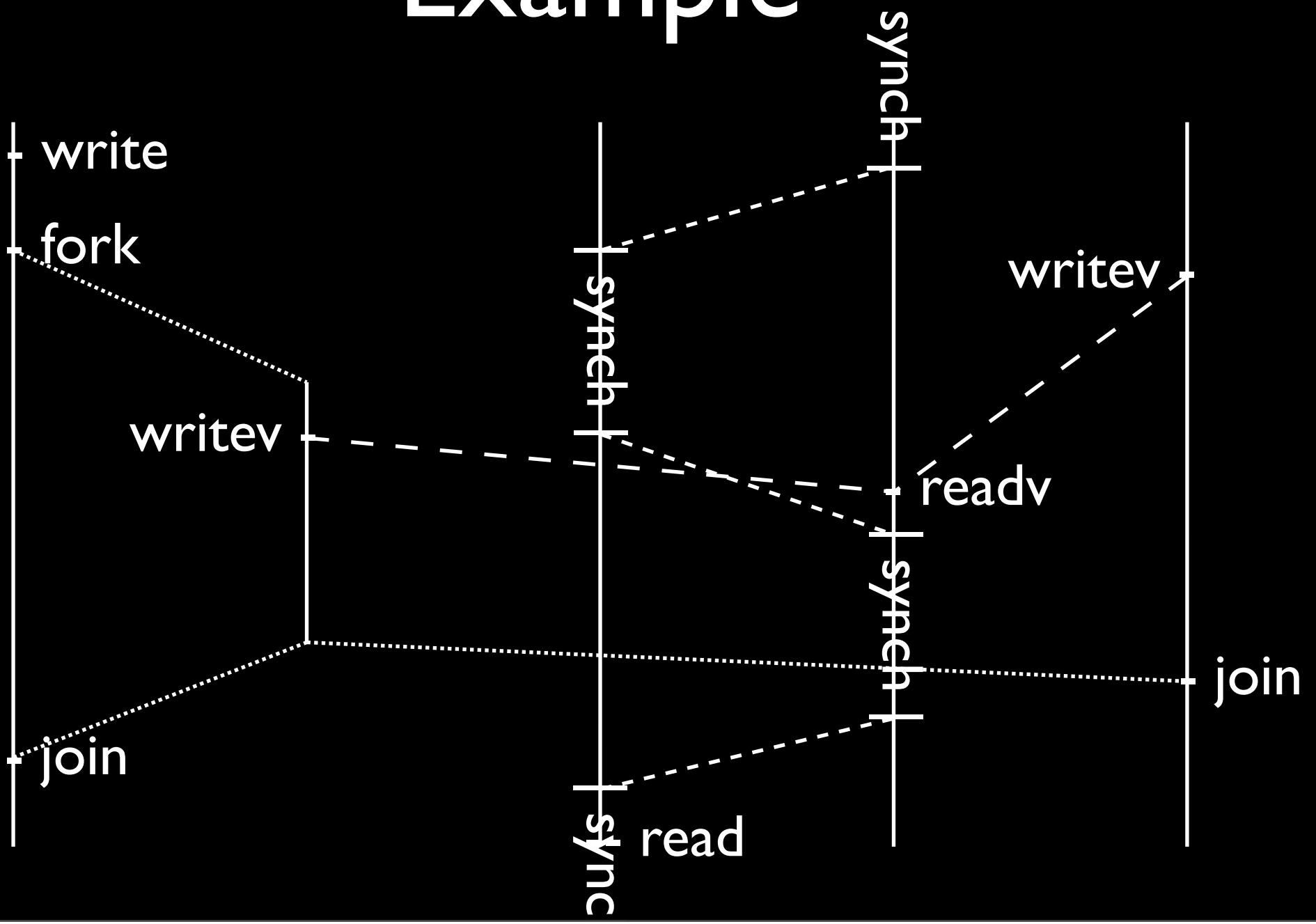
Example


write
fork
writev
join
sync
sync
sync
sync
writev
readv
join
read

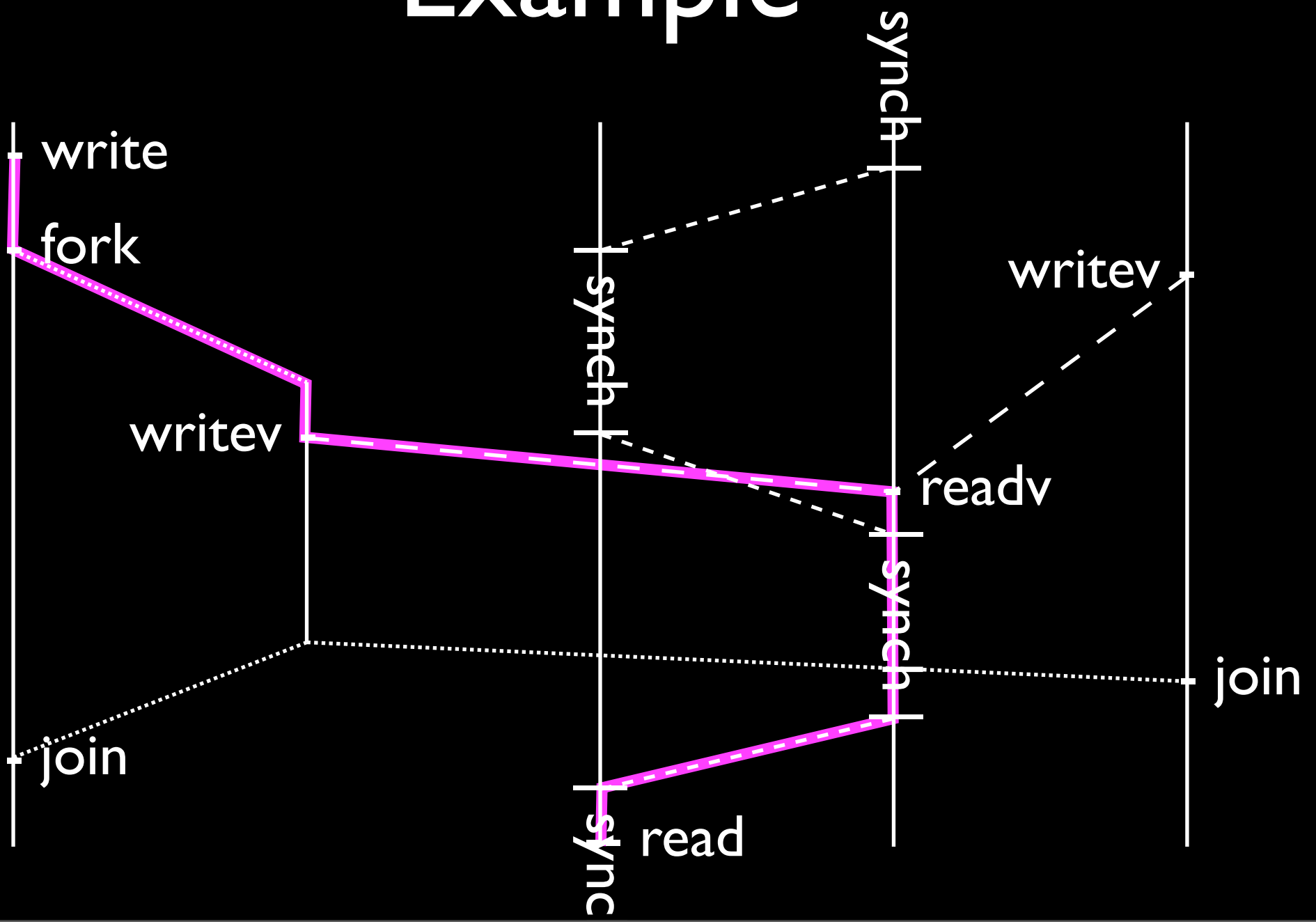
Example


write
fork
writev
join
sync
sync
sync
sync
writev
readv
join
read

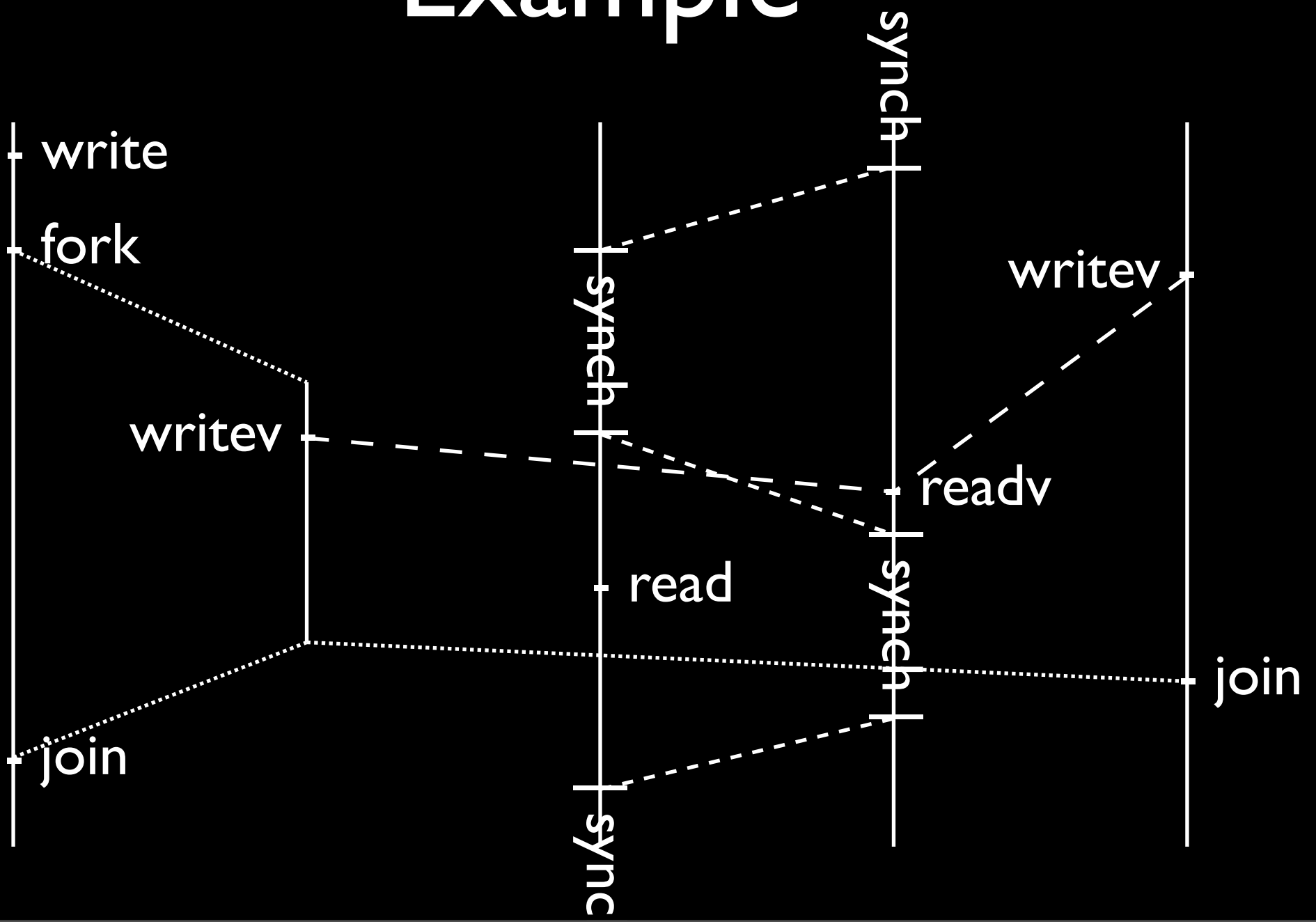
Example


write
fork
writev
join
sync
read
sync
writev
readv
sync
join

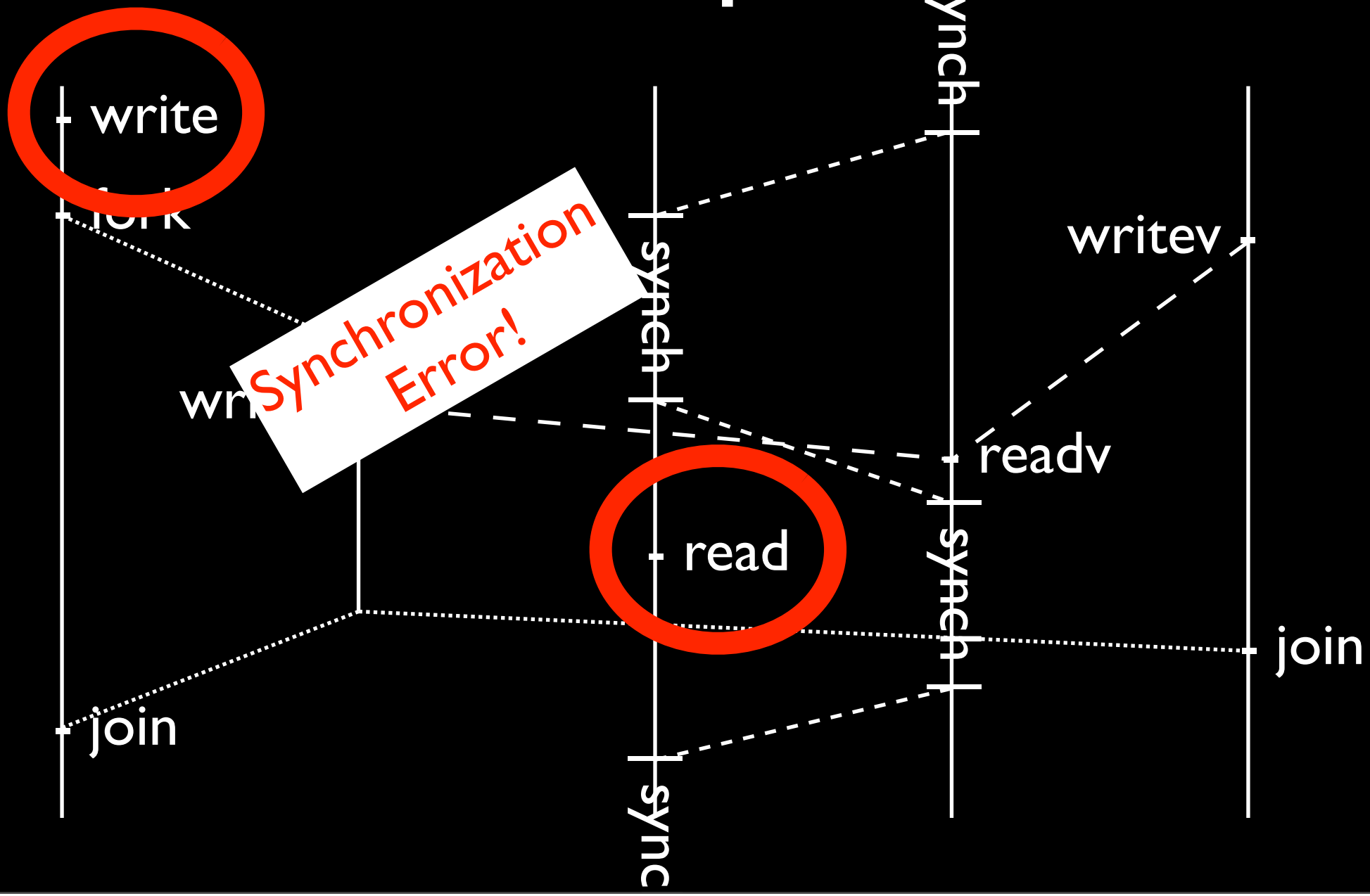
Example


write
fork
writev
join
sync
sync
read
sync
writev
readv
sync
join

Example


write
fork
writev
writev
readv
read
join
join
sync
sync
sync
sync

Example


write
fork
sync
writev
Synchronization Error!
wr
sync
readv
read
sync
join
join
sync

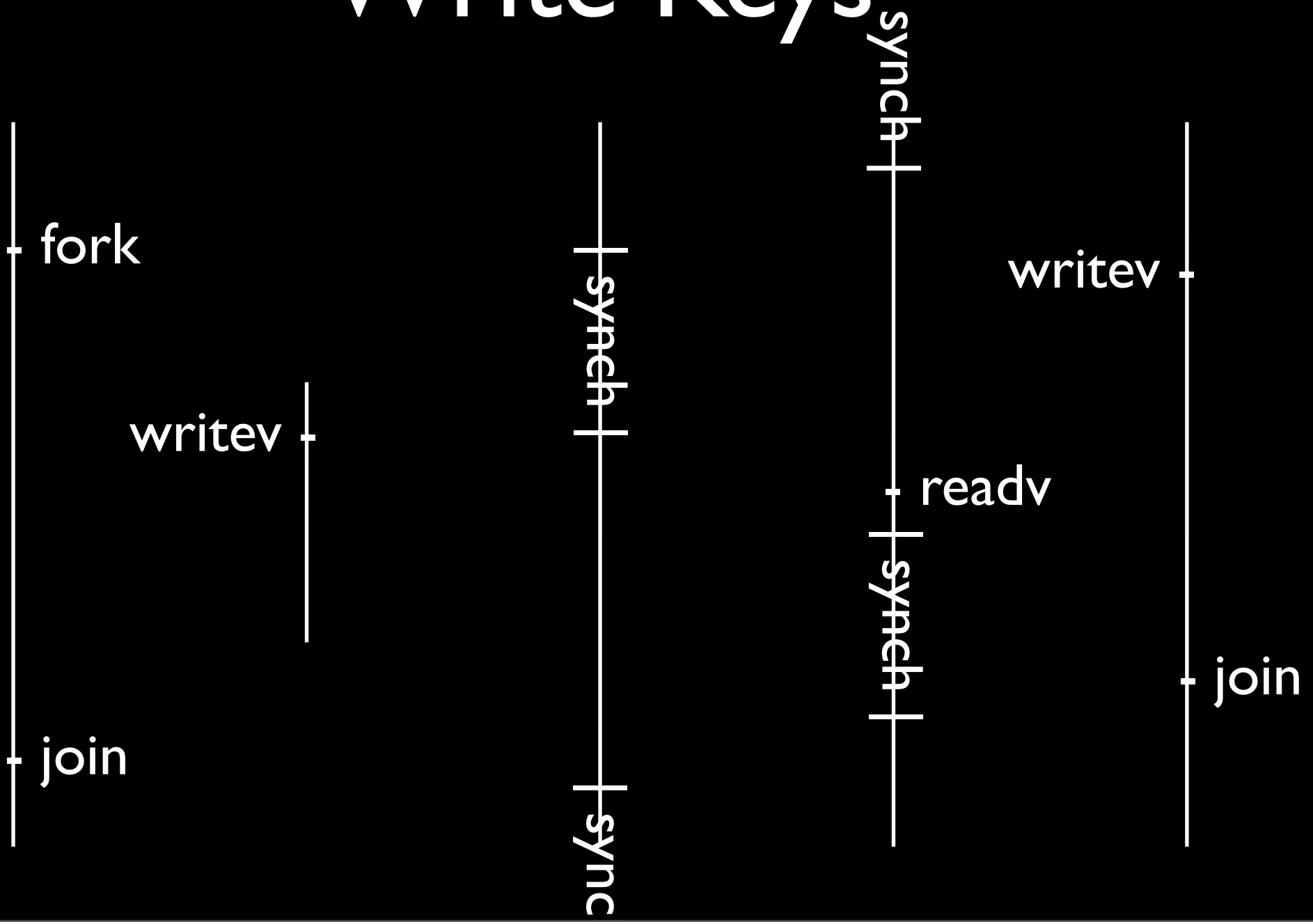
New Semantics

1. Start with a conventional store semantics;
2. Add concept of "write keys":
   - Every thread knows some keys (knowledge never lost);
   - New keys generated at writes;
   - Keys transferred through memory;
3. Knowledge required for access.

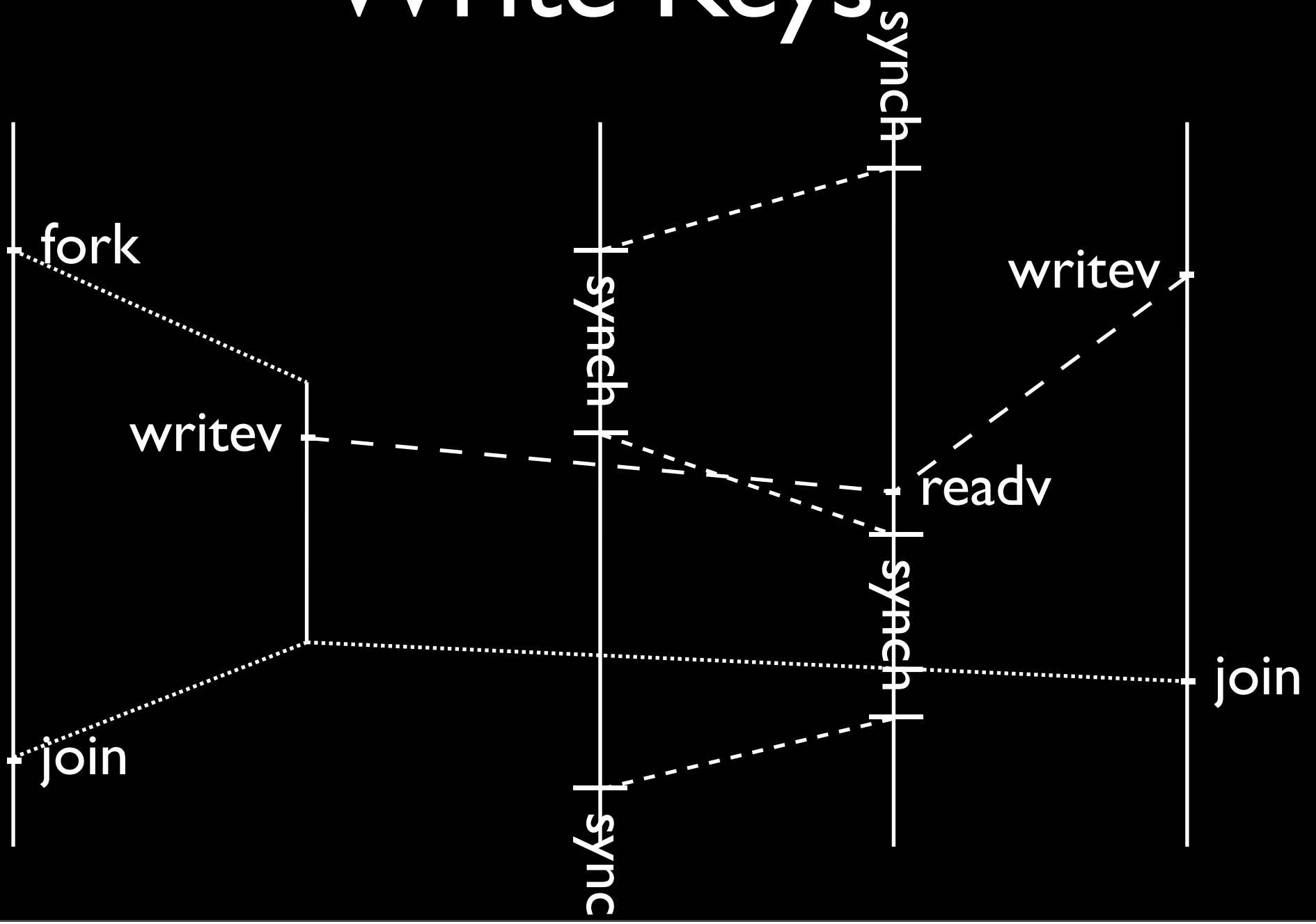
Simulate "happens before"

1. `fork` passes keys to new thread;
2. `join` picks up keys from thread;
3. `release` stores keys in mutex,
   `acquire` picks up keys from mutex;
4. volatile write adds keys to field,
   volatile read picks up keys from field.

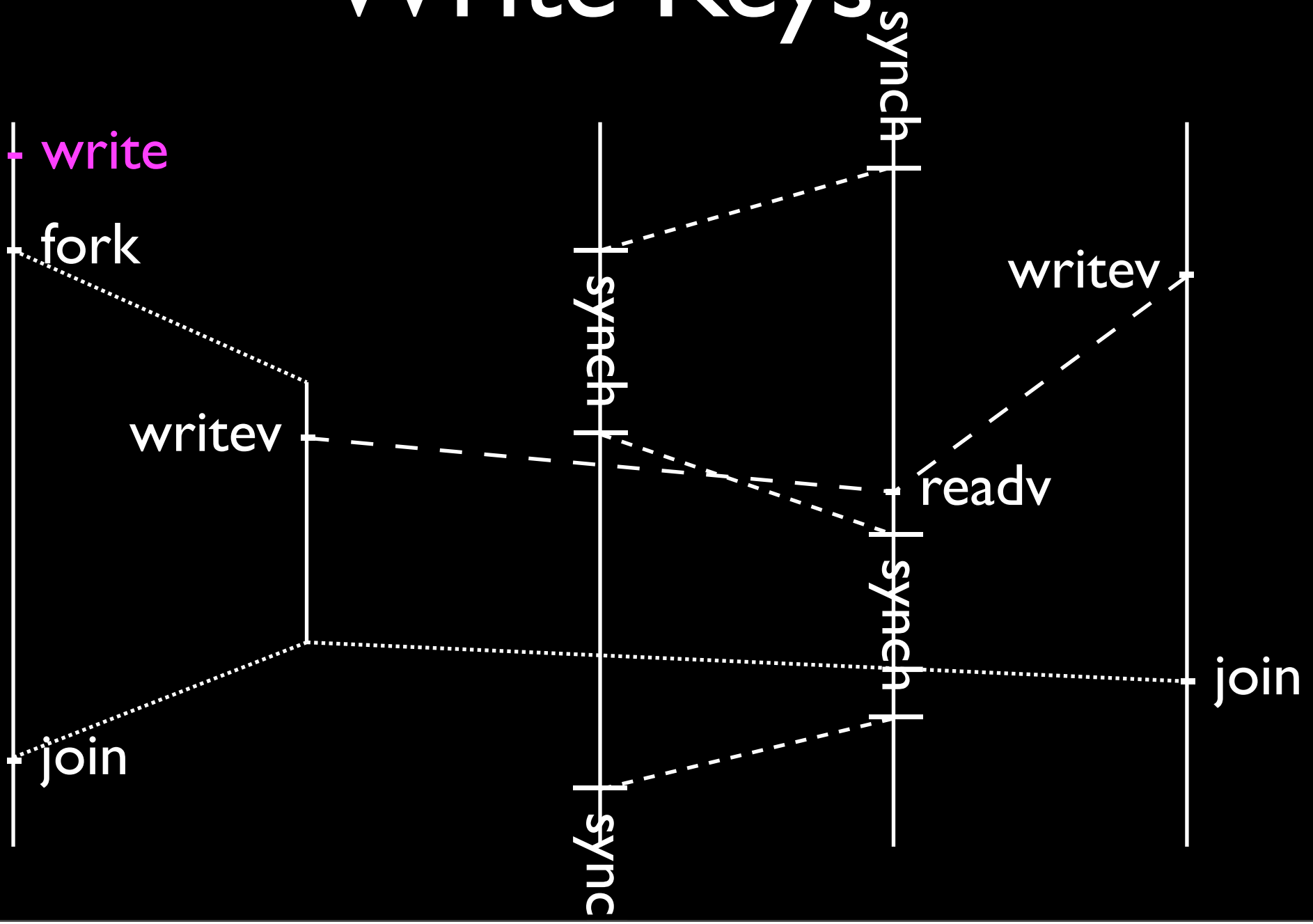
Write Keys



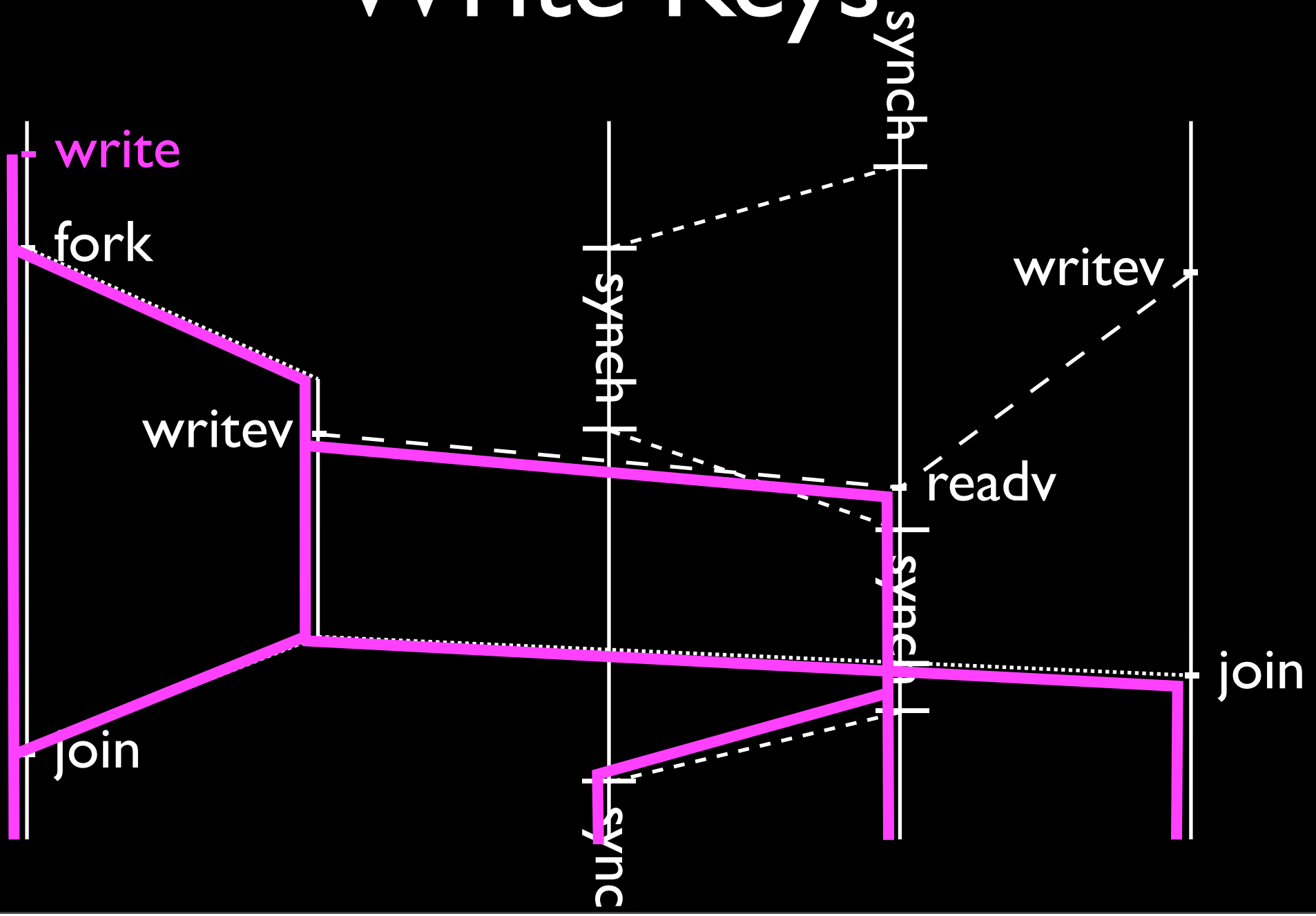
Write Keys



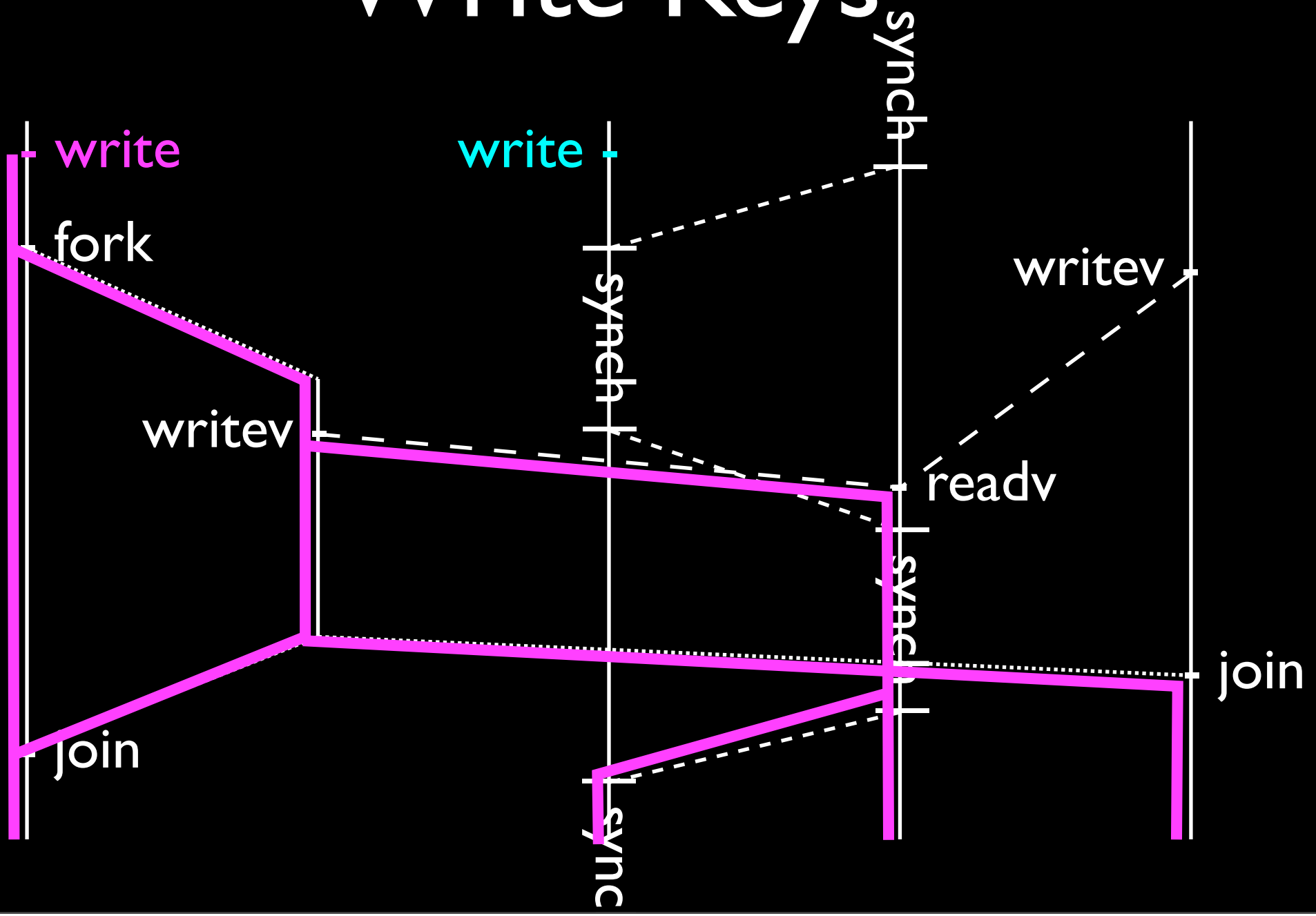
Write Keys


write
fork
writev
writev
readv
join
join
sync
sync
sync
sync

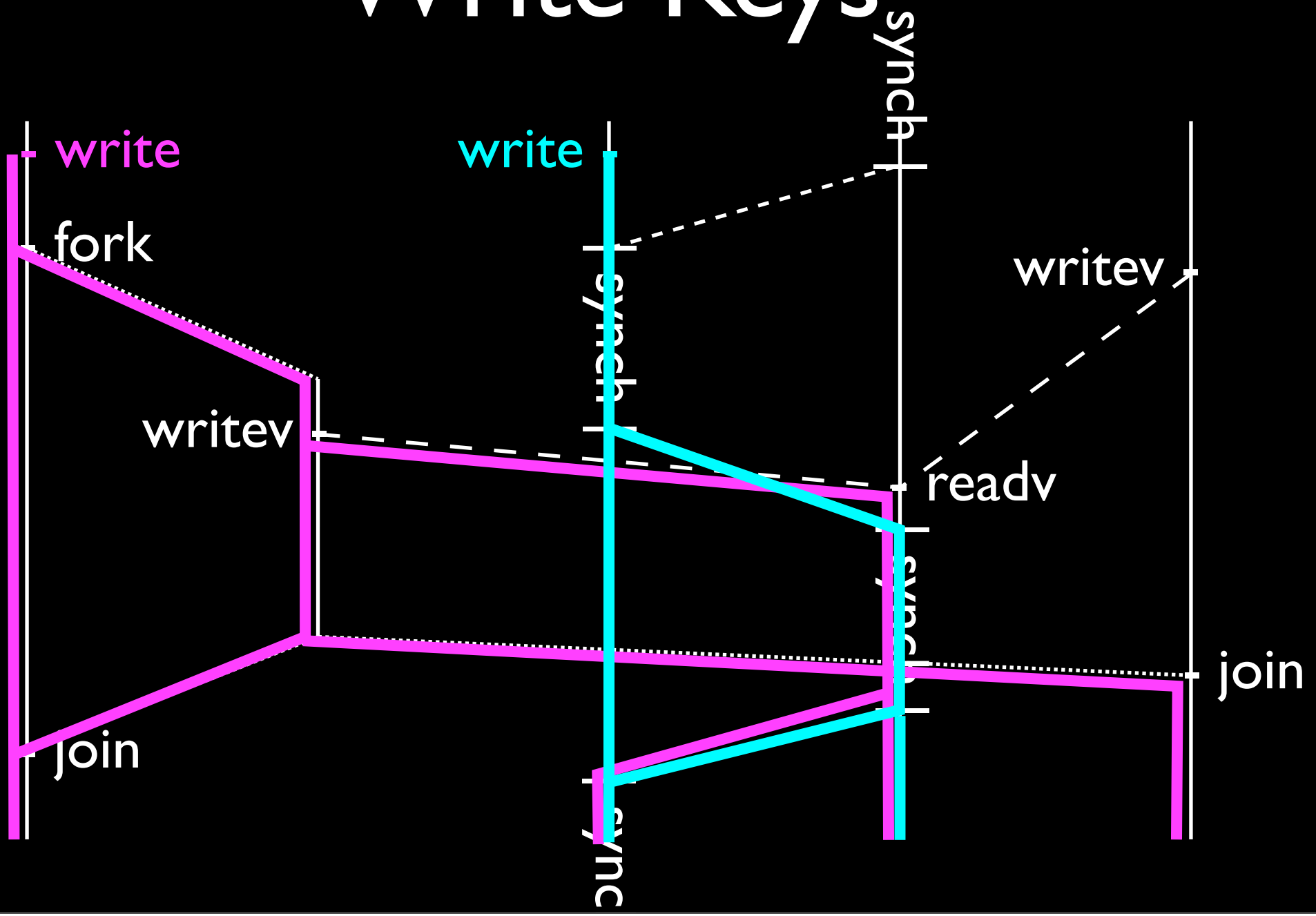
Write Keys


write
fork
writev
writev
readv
join
join
sync
sync
sync
sync

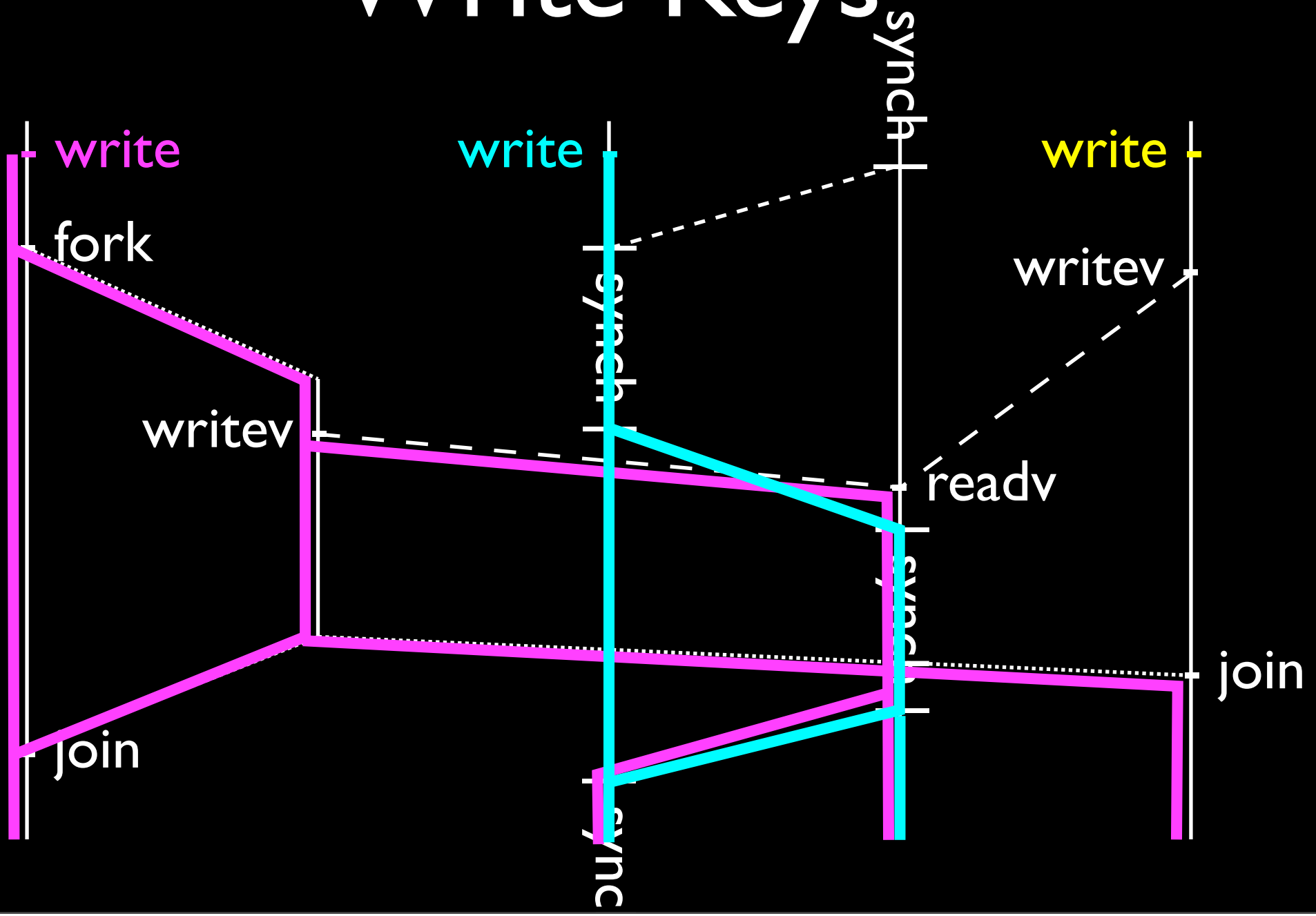
Write Keys


write
write
fsync
fork
fsync
writev
writev
readv
fsync
join
join
fsync

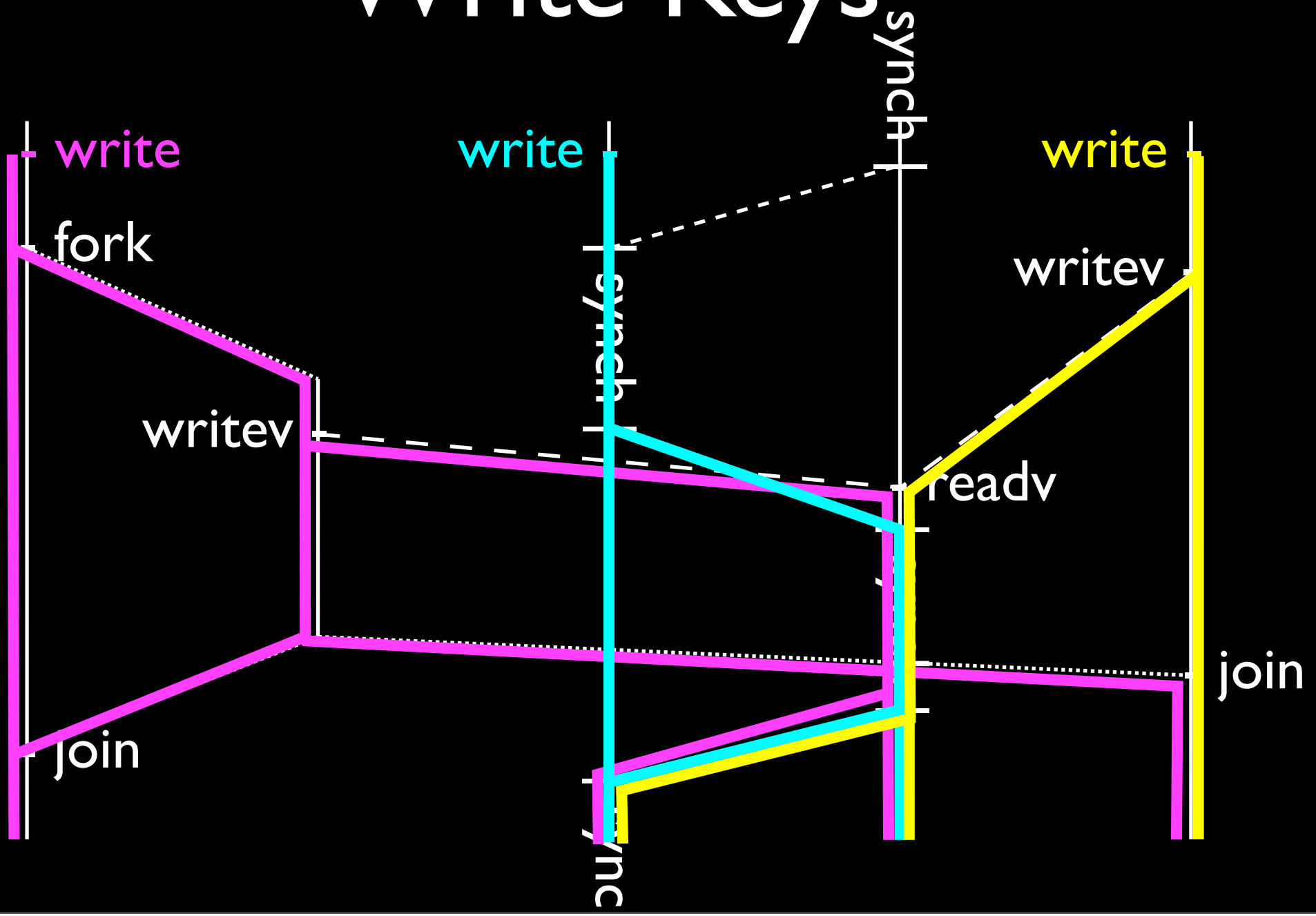
Write Keys


write
write
fork
writev
writev
readv
join
join
sync
sync
sync
sync

Write Keys


write
write
write
fork
writev
writev
readv
join
join
sync
sync
sync
sync

Write Keys


write
write
write
fork
writev
writev
readv
join
join
sync
sync

Write-Key Errors

- A thread is ready to access a field (either a read or a write);
- The write key for this field is some w;
- The thread does not know w;
- The thread blocks.

Theorem

The following three statements about a program are equivalent:

1. The program never has a write key error;
2. The program is correctly synchronized;
3. The program has no race conditions.

(Proved in Twelf.)

What is missing

- No guarantee that race conditions will be detected (in a particular run);
- No JMM-compliant semantics of incorrectly synchronized programs;
- No `wait`; no primitives; no dynamic dispatch; ...
- No type system.

Related Work #2

- Java Memory Model [Manson, Pugh ...]
- Goldilocks [Elmas, Qadeer, Tasiran 2007] Java implementation stores info approx dual to our write keys.
- Boehm and Adve [PLDI 2008] prove that programs using their C++ MM are correctly synchronized iff they have no races.

Applications

- Pre-Proved Idioms

  "If you follow these rules, your fragment will be accepted"

- Idiom Checkers

  Insulate programmer from theorem prover

- Pre-proved Scala traits accepted by Silicon.

Conclusions

1. Separate Proof Layers
   - Insulate programmers from low levels
   - Separate proofs, separate checking
2. Separation aided by a sound type system, at whatever level you are.

Questions?

- **Email:** boyland@uwm.edu

Safe Compound (vol.)

```
class UsingVolatile {
  private volatile CompoundData base;
  public void mutate() {
    synchronized (this) {
      base = base.clone().mutate();
  } }
  public int compute() {
    return base.compute();
} }
```

Safe Compound (vol.)

```
class UsingVolatile {
  private volatile CompoundData base;
  public void mutate() {
    synchronized (this) {
      base = base.clone().mutate();
  } }
  public int compute() {
    return base.compute();
} }
```

Not synchronized!

E-Write

$$\frac{\begin{array}{c}\mu(o.f) = (\{w\}, \_) \\ w \in \kappa(p) \qquad f \notin F_V \qquad w' \text{ arbitrary} \\ \mu' = \mu[o.f \mapsto (\{w'\}, o')] \qquad \kappa' = \kappa[p \overset{\cup}{\mapsto} \{w'\}]\end{array}}{(\mu; \theta; \kappa; o\,.\,f := o') \xrightarrow[g]{p} (\mu'; \theta; \kappa'; o')}$$

E-Write

$$\frac{\begin{array}{c}\mu(o.f) = (\{w\}, \_) \\ w \in \kappa(p) \qquad f \notin F_V \qquad w' \text{ arbitrary} \\ \mu' = \mu[o.f \mapsto (\{w'\}, o')] \qquad \kappa' = \kappa[p \overset{\cup}{\mapsto} \{w'\}]\end{array}}{(\mu; \theta; \kappa; o.f := o') \xrightarrow[g]{p} (\mu'; \theta; \kappa'; o')}$$

Thread *p* performs a write.

E-Write

$$\frac{\mu(o.f) = (\{w\}, \_) \quad w \in \kappa(p) \quad f \notin F_V \quad w' \text{ arbitrary} \quad \mu' = \mu[o.f \mapsto (\{w'\}, o')] \quad \kappa' = \kappa[p \overset{\cup}{\mapsto} \{w'\}]}{(\mu; \theta; \kappa; o.f := o') \xrightarrow[g]{p} (\mu'; \theta; \kappa'; o')}$$

Field Store "memory"

Thread p performs a write.

E-Write

$$\frac{\mu(o.f) = (\{w\}, \_) \quad w \in \kappa(p) \quad f \notin F_V \quad w' \text{ arbitrary} \quad \mu' = \mu[o.f \mapsto (\{w'\}, o')] \quad \kappa' = \kappa[p \overset{\cup}{\mapsto} \{w'\}]}{(\mu; \theta; \kappa; o.f := o') \xrightarrow[g]{p} (\mu'; \theta; \kappa'; o')}$$

Field Store "memory"

Known write keys

Thread p performs a write.

E-Write

$$\frac{\mu(o.f) = (\{w\}, \_) \quad w \in \kappa(p) \quad f \notin F_V \quad w' \text{ arbitrary} \quad \mu' = \mu[o.f \mapsto (\{w'\}, o')] \quad \kappa' = \kappa[p \overset{\cup}{\mapsto} \{w'\}]}{(\mu; \theta; \kappa; o.f := o') \xrightarrow[g]{p} (\mu'; \theta; \kappa'; o')}$$

Field Store "memory"

Known write keys

Thread p performs a write.

(non volatile)

E-Write

$$\frac{\mu(o.f) = (\{w\}, \_) \qquad w \in \kappa(p) \qquad f \notin F_V \qquad w' \text{ arbitrary} \qquad \mu' = \mu[o.f \mapsto (\{w'\}, o')] \qquad \kappa' = \kappa[p \overset{\cup}{\mapsto} \{w'\}]}{(\mu; \theta; \kappa; o.f := o') \xrightarrow[g]{p} (\mu'; \theta; \kappa'; o')}$$

Field Store "memory"

Known write keys

Field's current write key is w.

E-Write

$$\frac{\mu(o.f) = (\{w\}, \_) \qquad w \in \kappa(p) \qquad f \notin F_V \qquad w' \text{ arbitrary} \qquad \mu' = \mu[o.f \mapsto (\{w'\}, o')] \qquad \kappa' = \kappa[p \overset{\cup}{\mapsto} \{w'\}]}{(\mu; \theta; \kappa; o.f := o') \xrightarrow[g]{p} (\mu'; \theta; \kappa'; o')}$$

Field Store "memory"

Known write keys

Field's current write key is w.

(which thread p knows)

E-Write

$$\frac{\mu(o.f) = (\{w\}, \_) \quad w \in \kappa(p) \quad f \notin F_V \quad w' \text{ arbitrary} \quad \boxed{\mu' = \mu[o.f \mapsto (\{w'\}, o')]} \quad \kappa' = \kappa[p \overset{\cup}{\mapsto} \{w'\}]}{(\mu; \theta; \kappa; o\,.\,f := o') \xrightarrow[g]{p} (\mu'; \theta; \kappa'; o')}$$

Memory updated with new write key and value.

E-Write

$$\frac{\mu(o.f) = (\{w\}, \_) \quad w \in \kappa(p) \quad f \notin F_V \quad \mu' = \mu[o.f \mapsto (\{w'\}, o')] \quad w' \text{ arbitrary} \quad \kappa' = \kappa[p \overset{\cup}{\mapsto} \{w'\}]}{(\mu; \theta; \kappa; o.f := o') \xrightarrow[g]{p} (\mu'; \theta; \kappa'; o')}$$

Memory updated with new write key and value.

(which may be one no thread knows)

E-Write

$$\frac{\mu(o.f) = (\{w\}, \_) \quad w \in \kappa(p) \quad f \notin F_V \quad \mu' = \mu[o.f \mapsto (\{w'\}, o')] \quad w' \text{ arbitrary} \quad \kappa' = \kappa[p \overset{\cup}{\mapsto} \{w'\}]}{(\mu; \theta; \kappa; o.f := o') \xrightarrow[g]{p} (\mu'; \theta; \kappa'; o')}$$

Memory updated with new write key and value.

Thread p now knows the new key.